RESOURCE PROVISIONING IN LARGE-SCALE SELF-ORGANIZING DISTRIBUTED SYSTEMS

DISSERTATION

M. Brent Reynolds, U.S. Navy, Civilian

AFIT/DCS/ENG/12-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Navy, United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the United States Government and is not subject to copyright protection in the United States.

AFIT/DCS/ENG/12-03

RESOURCE PROVISIONING IN LARGE-SCALE SELF-ORGANIZING DISTRIBUTED SYSTEMS

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

M. Brent Reynolds, BA, MS

Civilian, USN

June 2012

AFIT/DCS/ENG/12-03

RESOURCE PROVISIONING IN LARGE-SCALE SELF-ORGANIZING DISTRIBUTED SYSTEMS

M. Brent Reynolds, BA, MS
Civilian, USN

Approved:

__// SIGNED //________________________ 15 MAY 2012
Kenneth M. Hopkinson, Ph.D. (Chairman) Date

__// SIGNED //________________________ 15 MAY 2012
Mark E. Oxley, Ph.D. (Member) Date

__// SIGNED //________________________ 15 MAY 2012
Barry E. Mullins, Ph.D. (Member) Date

Signed:

__// SIGNED //________________________ 04 JUN 2012
M. U. THOMAS Date
Dean, Graduate School of Engineering and Management

AFIT/DCS/ENG/12-03

Abstract

This dissertation presents on-line solutions to the challenge of scheduling large numbers of web services on significant numbers of servers. The services' resource demands change over time in response to surges and lapses in demand. The servers' resource supplies change dynamically to both expected and unexpected availability such as scheduled maintenance and network faults. This work presents a decentralized numeric method for quickly obtaining a solution to assign services to servers. Unlike other existing methods, this approach allows service availability and performance policy to be implemented through tunable parameters. The method is further enhanced by the use of control theory. Control is applied to the system's perceived demand of the services in response to their performance. By inflating the perceived demand of the service as a reaction to poor response time, services are reassigned and performance is shown to improve substantially. The research in this dissertation is unique because the services' demand and servers' resources are not considered fixed and the services are allowed to be reassigned.

AFIT/DCS/ENG/12-03

This work is dedicated to my maternal grandfather whose advocacy has never left me.

Acknowledgments

I acknowledge with great love and respect the patience and support provided by my children, parents, and closest friends. Thank you for your support, work, help, patience, advice, and love. This effort could not have been accomplished without you.

M. Brent Reynolds

Table of Contents

Page

Abstract iv

Acknowledgments vii

Table of Contents vii

List of Figures ix

List of Tables xi

List of Symbols xi

List of Acronyms xi

I. Introduction 1

II. Context 14

Introduction 14
Static, Dynamic, On-Line 14
Representation Layer 19
Quantification Layer 20
Qualification Layer 24
Organization Layer 27
Summary 28

III. Testbed 30

Introduction 30
Physical Architecture 30
Load 32
Services 34
Summary 46

III. Provisioning Norm: A Placement Quality Measure to Balance Service Performance and Inclusion 50

Introduction 50
Quantification 52
Service Performance and Inclusion 60
Empirical Methodology 79
Other Measures 92
Heuristic 96

Related Works.........103
Summary.........107

V. On-Line Service Profiling, Self-Organization, and Caching Configurations.........109

Introduction.........109
Related Works.........112
Profiles, Models, and Measures.........115
Self-Organization.........118
A Simple Experiment.........123
Results for the Simple Experiment.........126
Tenant Placement and Caching.........129
Theoretical Analysis.........130
Experimentation.........141
Results.........145
Cost of Change.........145
Quality.........148
Response Time.........152
Summary.........156

VI. Performance Feedback Loops in Self-Organizing Web Servers.........157

Introduction.........157
Related Works.........159
Methodology.........162
Theoretical Basis.........171
Empirical Results.........186
Summary.........194

VII. Conclusions and Recommendations.........196

VIII. Bibliography.........201

List of Figures

Figure Page

1. Service Abstraction Stack 3

2. Cloud Chamber Architecture 4

3. Cloud Chamber Architecture 31

4. Inside the TinyCore Node 33

5. Database Entity Relationship Diagram 41

6. Services Included with Low Resource Demand 71

7. Service Included with Low-Medium Resource Demand.................72

8. Services Included with Medium Resource Demand........................73

9. Services Included with High Resource Demand 74

10. Histogram of 10 Million Sample Configuration Provisioning Norms.........82

11. Heuristic's Decent to the Best Quality...84

12. Performance Plotted Against Worsening Analytical Quality 86

13. Services Included and Response Time Plotted Across α..............................89

14. Services Included as Ratio of Supply to Demand Increases..........................90

15. Kullback-Leibler and Provisioning Norm Inclusion...................................96

16. Heuristic Formally Defined 99

17. Predefined Traffic for Simple Experiment 126

18. Overall Mean Response Times for the Simple Experiment.......................127

19. A Two-Dimensional Representation of Configuration Space 132

20. Mean Traffic Across All Services Over Thirty Minutes 142

21. Cumulative Cost of Change 147

22. Configuration Quality with Random 150

23. Configuration Quality with Mixed...151

24. Configuration Quality with Caching Only..152

25. Mean Response Time for Random 153

26. Mean Response Time for Mixed...154

27. Mean Response Time for Caching Only...155

28. Configuration Regions 183

29. Mean Response Times Over Time...192

30. Mean Services Included Over Time 193

31. Coefficients Over Time...194

List of Tables

Table Page

1. Resource Typology 23

2. Nodes SQL Database Table with IP and Profile...43

3. Services SQL Database Table with ID and URL (25 of 100 records).....................42

4. Loadrunners SQL Database Table with IP and Num (25 of 100 records)...............45

5. Prescribed Load SQL Database Table (25 of 241,000 records) 46

6. .Sample Node and Service Resource Profile Matrices 81

7. Mean Response Times Over Increasing Load 91

8. Mean Services Included Over Increasing Load...91

9. Epoch Reconciliation & Configuration Over Time 122

10. Actual and Normalized Node Resource Profile Values (10 of the 25)..............123

11. Actual and Normalized Service Resource Profile Values (25 of the 100) 124

12. Experimental Cases: Cache Seed and Random Seed Combinations 145

13. Cost, Quality, and Response for All Random & Cached Seed Cases..................145

14. Small & Large δ Seeds: Cache vs. Random in Mixed Cases 148

15. Response Times with Varying Levels of Control..188

16 - Services Included with Varying Levels of Control (t>1800) 188

17. Hamming Distance with Varying Levels of Control (t>1800)............................189

18. Coefficient Values with Varying Levels of Control (t>1800, coeff>1) 189

19. Queue Size with Varying Levels of Control (size>0 and t>1800) 190

20. Queue Size with Varying Levels of Control (size=0 and t>1800) 190

List of Symbols

Symbol	Meaning
$\sum$	Summation
$\cup$	Union
∞	Infinity
$\lVert\bullet\rVert_1$	Matrix 1-norm
$\lVert\bullet\rVert_F$	Matrix Frobenius Norm
$\lVert\bullet\rVert_{\alpha,F}$	Provisioning Norm
$\lvert\cdot\rvert$	Absolute value or set cardinality
$\leftarrow$	Assignment operator
$\lceil\ \rceil$	Ceiling operator
$\Leftrightarrow$	bi-implication
$\leq$	Less than or equal to
$\geq$	Greater than or equal to
$\otimes$	Kronecker product
$\odot$	Hadamard product
$\vec{v}\blacklozenge M$	$\vec{v}^T \otimes \mathbf{1}^T \odot M$
M_-	Matrix of non-positive entries in M
M_+	Matrix of non-negative entries in M
$\mathbf{0}$	Matrix of zeros

$\mathbf{1}$	Vector of ones
α	Inclusion Parameter to Provisioning Norm
α	(Chapter 2)Competitiveness of Algorithm A
b_i	Profit of item in knapsack
$\mathbb{C}^{+}_{N,S}$	$\left\{C \in \mathbb{C} : \forall ij, \left[N - CS\right]_{ij} \geq 0\right\}$
$\mathbb{C}^{-}_{N,S}$	$\left\{C \in \mathbb{C} : \exists ij, \left[N - CS\right]_{ij} < 0\right\}$
C_r	An arbitrary configuration
C_c	The current configuration
$d()$	Hamming Distance
δ	Matrix difference between matrices
Δ	Set of δ
$\vec{\Delta}_s$	Service adjustment vector
ε	Smallest unit of a resource (resolution)
$E\left[X\right]$	Expected Value of X
$e_i\left(t\right)$	Control error at time t
$\in$	Element operator
$\mathcal{F}$	Set of functions
h	heuristic
i	index or set element
K_p	Zeigler-Nichols proportional coefficient

min	Minimum value of a set or range
max	Maximum value of a set or range
μ	Mean
N	Node matrix
N^{*}	Effective node matrix
$\boldsymbol{N}$	Poser set of *N*
n	Number of nodes or a specific node
$\vec{n}$	Node profile vector
neg(x)	min{x,0}
$q(\ )$	Quality function
Q	Quality function
pos(x)	max{x,0}
$P(X)$	Probability of event *X*
$\mathcal{P}(\ )$	Power set operator
ρ	Sequence of inputs to on-line algorithm
$\mathbb{R}^{+}$	Non-negative real values
r	Number of resources
S	Service matrix
S	(Chapter 2) Set of states
S^{*}	Effective service matrix
$S^{(n)}$	Services assigned to node *n*

$\boldsymbol{S}$	Power set of *S*
$\boldsymbol{S}_{\mathbf{n}}$	Services assigned to Node *n*
$s(i)$	(Chapter 2) ith state
s	Number of services or a specific service
$\vec{s}$	Service profile vector
sup	Supreme mum
σ^2	Standard Deviation
t	Time step
T	(Chapter 2) Task
T_i	Zeigler-Nichols integral coefficient
T_d	Zeigler-Nichols derivative coefficient
W	knapsack weight capacity
w_i	Weight of item in knapsack

List of Acronyms

Acronym	Meaning
AF	Air Force
APP	Application Placement Problem
BA	Bachelor of Arts
CD	Compact Disc
CGI	Common Gateway Interface
CMMI	Capability Maturity Model Integration
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
FIFO	First In First Out
Gb	Gigabit
GB	Gigabytes
GHz	Gigahertz
HP	Hewlett-Packard
http	Hyper Text Transfer Protocol
IO	Input Output
IP	Internet Protocol
ISO	International Organization for Standardization
JSDL	Job Submission Description Language
KB	Kilobytes
KL	Kullback-Leibler

KLD	Kullback-Leibler Distance
LRU	Least Recently Used
Mb	Megabit
Mbps	Megabits per second
MB	Megaabytes
MHz	Megahertz
MIMO	Kullback-Leibler Distance
MMKP	Multi-Dimensional Knapsack Problem
ms	milliseconds
MS	Master of Science
OSI	Open Systems Interconnection
OTPP	On-Line Tenant Placement Problem
PaaS	Platform as a Service
PID	Proportional Integral Derivative
POMDP	Partially Observable Markov Decision Process
QoS	Quality of Service
RFC	Request For Comment
SaaS	Software as a Service
SNMP	Simple Network Management Protocol
SLA	Service Level Agreements
SQL	Structured Query Language
TCP	Transport Control Protocol
TDH	Tenant Dispatch Heuristic

TTL	Time To Live
UDP	User Datagram Protocol
URL	Universal Resource Locator
USB	Universal Serial Bus
USAF	United States Air Force
USN	United States Navy
XML	Extensible Markup Language

RESOURCE PROVISIONING IN LARGE-SCALE SELF-ORGANIZING DISTRIBUTED SYSTEMS

I. Introduction

Cloud computing, a form of large-scale distributed computing, has expanded in availability, affordability, and reliability. Private providers such as Google, Microsoft, Amazon, and Apple have expansive server farms spread across the globe. Other institutions, including the government and military, have consolidated computing resources into large data centers. These facilities collectively present millions of computing resources to individuals and organizations. Due to scale, competition, and advertising revenues, services such as email, social networking, office document processing, file storage and backup, banking, music, and retail are free or near free.

Millions of individual services execute on millions of computing resources. The organization and administration of these services is a logistical challenge. Currently, two independent approaches address this challenge. The first dedicates a physical computing resource to each service or bundle of services. This approach, for the sake of performance, wastes available computing resources, electricity, and physical space. Purchasing cloud computing by the CPU hour uses this approach to dedicate physical resources. The second approach minimizes the number of physical machines by consolidating services on virtual machines placed on shared physical servers. This approach has been popular in corporate information technology departments in the recent past because of the savings in administrative costs such as hardware expense and electricity. This consolidation approach can either over-estimate the requirements for each service, thus wasting resources, or under-estimate the requirements for each service, thus jeopardizing performance goals.

This dissertation assumes two driving forces. Over time, more services will be demanded and the speed and capacities of physical computing servers will increase. These two assumptions imply more services will be placed together on the same physical computing servers in the future. The administrative challenge will become more complex and larger in scale. Manual human intervention will be insufficient and too expensive. Autonomous management methods must be employed to keep up with service level demands and the growing scale of the operations. The servers must self-monitor and self-administer within the policy bounds defined by the provider. Furthermore, services' resource consumption must be continuously and accurately profiled such that quality autonomic decisions align with the policy goals. This dissertation explores this large scale challenge by presenting theoretical and empirical results of efficient, autonomic, policy driven service resource management.

To understand where this work fits into the larger picture, Figure 1 shows a partial generalization of the architecture laid out by Chang, et al. [1]. Layer 0 is the physical servers providing simple physical processing and storage. Layer 1 consists of the virtualized instances of operating system services. Layer 2 consists of the database/web/middleware executable instances installed and running on the virtual server in the layer below. Layer 3 abstracts the executable instances to the consumer. For example, a particular web service may run on a dozen actual web servers while the consumers of the web service are unaware of the load balancing.

This work resides in the Layer 3 Provisioning Abstraction Layer. Similar to the consumer's experience with load balancing, the publisher of a web service should interact with the provisioning layer and be unconcerned with the working of the layers below, as long as the expected service level agreements continue to be met. Similarly, the administrator should be able to "turn the dials" of policy at the provision layer with the expectation the system will adjust to the new policy while maintaining compliance with all previous policies and service level agreements. As outlined in [1], the lower layers of abstraction react not only to the work load demands and the system administrator but also to environmental conditions such as system failures and security breaches. The aim of this research is to measure the services and resources, to evaluate the quality of a configuration, to quickly calculate (on a large scale) configurations in which services are not under-resourced, to utilize performance feedback to find well performing configurations, and finally, to apply administrative policies to the method.

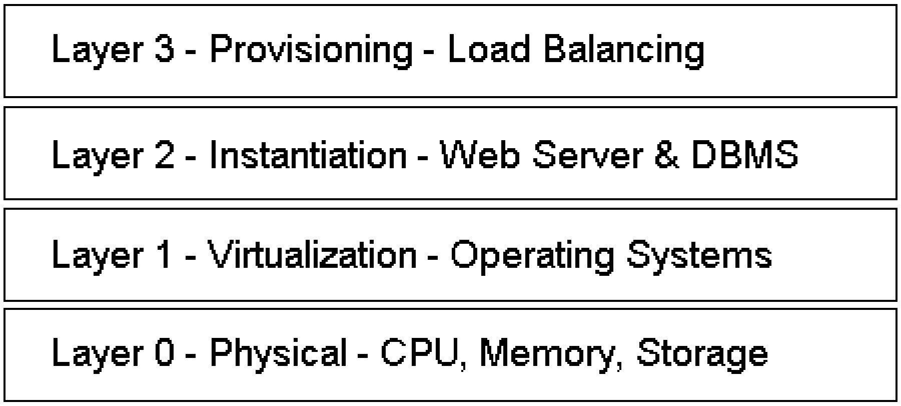

Layer 3 - Provisioning - Load Balancing
Layer 2 - Instantiation - Web Server & DBMS
Layer 1 - Virtualization - Operating Systems
Layer 0 - Physical - CPU, Memory, Storage

Figure 1. Service Abstraction Stack.

In Microsoft's Azure cloud architecture utilizes what they call the Fabric Controller, see Figure 2. This Fabric Controller is responsible for allocating virtual machines and assigning work to the virtual machines. As demand ebbs and flows, the Fabric manages the load. According to Chappell [56], "This magic is performed by the fabric controller, a fundamental aspect of Windows Azure." As of the writing of this dissertation no documentation is made available on the workings of this Fabric Controller. This dissertation addresses challenges encountered by mechanisms like the Fabric Controller.

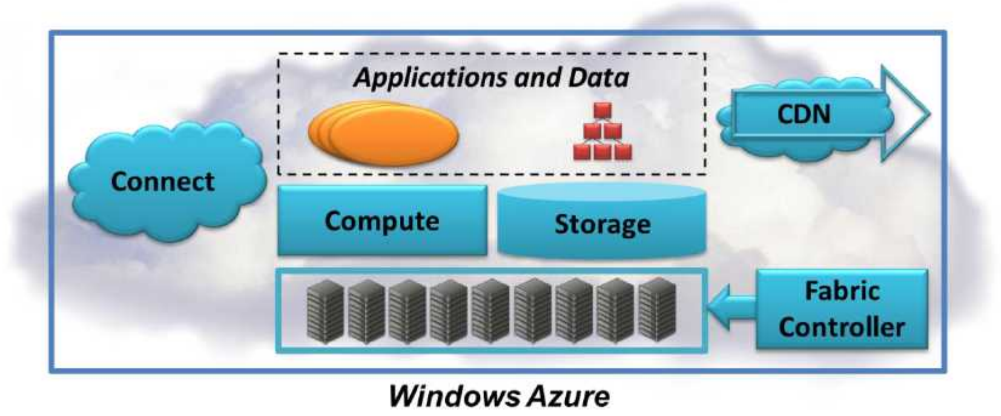


Figure 2. Microsoft Azure Achitecture.

The work of Kephart [2] outlines the research challenges in autonomic computing. Kephart breaks the autonomic research space into three major considerations: autonomic elements, autonomic systems, and human computer interactions. Kephart further refines each of these into sub-categories and addresses each of them in detail outlining an ambitious list of research needs in a wide variety of areas such as multi-agent behavior, operations research, systems engineering, software engineering, economics, human-system interfaces, optimization, and machine learning. The following summarizes his challenges.

- Specific autonomic elements to improve the self-managing capability of web servers, database servers, etc.
- Generic autonomic element technologies: planning, modeling, forecasting, optimization, etc. of the above autonomic elements.
- Generic autonomic element architectures, tools, and prototypes to create autonomic elements.
- Autonomic system technologies to achieve system-level goals, including automated provisioning, workload management, and automated installation.
- Autonomic system architectures to govern interactions among autonomic elements.
- Autonomic system science research on fundamental science of large-scale autonomic computing systems.
- Human studies research interactions between human administrators and self-managing systems to determine the most effective interfaces.
- Policy research on high-level policies and appropriate transformation into autonomic systems behavior.

His list progresses from low level components to system level and to policy level. The theoretical model and testbed presented in this dissertation address each of these items with the exception of human interfaces. At the low level, the web servers are equipped with self-management technologies such as sensors and actuators to measure and move services as needed. The web servers communicate among themselves to share system wide conditions. Using mathematical algorithms the behavior of the system is governed to meet the human derived policies. These policies are implemented mathematically as parameters to the algorithms.

The research in this dissertation explores the theoretical and empirical requirements for several servers hosting services, which in concert monitor resource availability and continuously arranging these services in response to policy and conditions. While job scheduling and application placement have been discussed at length over the years, none of these models specifically address the new challenges presented by transactional web services in a large scale, highly virtualized environment. Previous works address individual aspects of the problem [3][4][5][6], but none in its entirety.

Classic job scheduling on computing machines is as old as modern computing. The classic problem revolves around how long the jobs take to run. A single dimension, usually process time or CPU utilization, is obtained *a priori* and assigned to the job. Jobs are either scheduled to optimize processing on a single server or scheduled simultaneously across multiple servers. Variations on this problem include maximizing throughput, minimizing cost, maximizing profit, maximizing utilization, minimizing number of machines; too many to list exhaustively. Static, dynamic, and on-line versions of the problem exist in the literature. More recent interest in job scheduling has concentrated on reducing power consumption by minimizing the number of physical machines. Virtualization pushes to fit as many virtual machines on as few physical machines as possible. As more virtual machines arrive, more physical machines are made available.

In these scheduling problems, regardless of the variation, jobs are assigned and never removed or, as in queuing theory, the job leaves after some amount of service time. Jobs are independent of each other. Jobs do not leave the system and return. This dissertation deviates from these classical problems and addresses web service scheduling such that the service change their resource requirements over time, persist over time, and are moved from one server to another over time.

This work creates a novel model for transactional web service performance and resource management: the On-Line Service Placement Problem. This problem is derived from the Application Placement Problem [6] and generalizes the On-Line Tenant Placement Problem [7]. The problem is how to effectively and efficiently assign the computational resources of a large scale network to the demands placed upon it by services deployed to the network *and* meet performance goals. This on-line, real-time, and on-going process adapts to environmental conditions such as node and service churn and the system administrator policy changes. First, the focus of the assignment must minimize the service level agreement violations, e.g. the requirements of the services must be met as much as possible. Second, the costs of changing the assignments must be minimized, e.g. moving a service from one node to the next is not free in terms of time and communication. Third, the amount of resources to meet the requirements should be minimized. This dissertation offers a novel theoretical and empirical approach to define and solve the problem.

Specifically, this work considers services that persist and change over time. Unlike previous job scheduling problems, the services are not removed and, once placed, change their resource demands. Services are not removed because users continue to consume them over time. The consumption varies over time as user needs change through popularity, diurnal cycles, or other market forces. The On-line Service Placement

Problem defined in this dissertation has the following requirements which make it a new and unique problem.

- Multiple resources considered simultaneously.
- Service resource requirements profiled on-line.
- Services moved for better fit.
- Service performance used as an input.
- Overloading allowed.
- Decentralized.

This model addresses new challenges presented in virtualized cloud computing environments. These challenges arise from the virtualized processing and scale. Virtualization creates a processing unit (virtual machine) capable of moving to different physical hosts and capable of sharing those physical resources. This processing unit's resources such as CPU, memory, and bandwidth can adjust to best meet the demands placed on it. Much research [8][9][10] has examined the placement of virtual machines on physical machines. Other research [11] examines the efficiency of capturing the virtual machines' state and effectively moving virtual machines from one physical machine to the next. Other research [12] shows how to add additional virtual machines (clones) as demand increases. Contention for shared resources by multiple virtual machines is studied in [13]. Commercial services such as Amazon and Microsoft offer these services now, offering as much scale as one can afford.

On these virtual machines, applications in the form of web services execute. These services are as sophisticated as streaming millions of movies to millions of users such as Netflix, which executes on Amazon's services. Social web sites such as Reddit use Amazon’s service to provide bulletin board and user discussions. Using Microsoft’s development tools, small and large customers code and deploy to Microsoft’s Azure service. Currently, these services provide virtual machines, billing by the CPU hour (lower than $0.10 per hour per virtual machine). For this service, a physical CPU is bound to the customer’s virtual machine. This guarantees performance and prevents virtual machines from contending for available resources. To re-iterate, the customer provides a virtual machine and purchases a physical CPU to execute the virtual machine.

This model creates two problems. First, the physical machines have resources wasted on idle time and operating system functions. The customer pays for wasted processing capacity. The vendor has expensive, under-utilized hardware. Second, the burden of operating system administration falls onto the customer. These duties include and are not limited to user management, permissions, licensing, security patches, and software updates. The customer incurs these additional costs. The vendor relies on the customer to properly manage and secure these systems, thus incurring additional costs to protect themselves from the virtual machines.

This dissertation asserts these two problems will drive the model to change. Applications will be developed and uploaded free of operating system administration. The deployed web services will execute in an environment with a web server to handle requests and an application server to execute the logic. These servers will properly isolate competing applications just as well as the virtual machines today. This dissertation does not address these security considerations of services sharing resources, such as malicious services and out of control services, and instead focuses on the problem of autonomously balancing resource utilization and performance.

The model proposed in this dissertation utilizes sensors to measure resources. These software based sensors measure the resource consumption of services and the resource supply of the servers (nodes). These CPU utilization, memory, and bandwidth measures create on-line profiles of the services and nodes. Each profile is a vector containing the numeric representation of the service demand and the node supply. In order to find a proper placement of the services, these profiles are the inputs to a vector packing problem [14] similar to multi-dimensional bin packing problems or multi-dimensional knapsack problems. However, the On-Line Service Placement Problem generalizes previous vector packing problems. Unlike previous vector packing work, once placed, a service can be moved, *and* the size of the packed services change size. This problem has not been addressed in the literature.

Assuming services do not have to be assigned and when assigned are to only a single node, the number of possible placement configurations of s services on n nodes is s^{n+1}. For hundreds of nodes and services this search space is quite large. Two aspects of this space are considered in solving this placement problem. The first aspect is how to compare one placement to another. The placement's quality is a partially ordered ranking from best to worst. This allows each of the s^{n+1} possible configuration to be given a rank to compare which is better. The second aspect is how to search through the large space of possible placements to find the better placements. Finding a better placement needs to be fast enough to be useful given that the service profiles change over time. The demands of the services can change quickly depending on user demand and other environmental conditions. The search must operate in seconds or minutes. Long running techniques such as Integer Programming and Brand and Bound can not be considered for such short turn around.

The *quality* of a placement maps each placement to a real value. This value allows for ranking and comparison of placements. The mapping of placements to real values is derived from a policy, i.e. what factor is important affects the magnitude of the value. A quality function could evenly balance the load of services across all nodes, could fill nodes to minimize the number of servers, or could favor placing services on nodes with similar profiles. These three policy examples all independently meet their goals when service demand is less or much less than the node supply, e.g. all the services easily fit onto the nodes per the policy. As demand levels approach supply levels, these policies leave unassigned those services that do not fit. These functions are therefore not capable of providing more generalized policies. This dissertation presents a quality function, the Provisioning Norm, allowing for the possible over-allocation of nodes' resources using a tunable policy parameter dialed to the tolerance for over-allocation. The Provisioning Norm provides policy as a mathematical balance between availability and performance. At one extreme, availability is critical. In this case, over-allocation of resources is essentially ignored, e.g. all services are available for consumption. At the other extreme, performance is critical. In this case, over-allocation of resources is prohibited, e.g. as needed services are excluded and unavailable. In every case in between these extremes, exclusion and over-allocation are inversely tolerated over the range of the tunable parameter. This allows a policy of performance and over-allocation to be expressed as a number. To this end, a numeric method, presented later, calculates the precise parameter value to include a specified number of services.

The next chapter defines an abstracted model to frame the discussion of the facets of this dissertation. The chapter discusses the nature of on-line problems as compared to static and dynamic problems. The abstracted model stratifies the various aspects of the problem into the physical layer, quantification layer, qualification layer, and the policy layer.

The empirical results discussed in Chapters 4, 5, and 6 were acquired using a custom built testbed. The testbed, described in Chapter 3, is a virtualized environment leveraging several technologies such as the TinyCore Linux operating system, Lighttpd web server, Microsoft's SQL Server, and three custom applications. These tools implement the theory and verify the expected results.

Chapter 4 is based on a paper [15] presented at the 2011 Service Computing Conference and a journal paper submitted to the journal Service Oriented Computing and Applications. In this chapter, mathematical theorems and models implement policy while predicting the policy's performance in a large scale environment. These formulas and numeric methods derive optimal parameter settings to predictably balance service performance and service availability of the static problem.

Chapter 5 is based on a paper [16] presented at the 2012 IEEE Hawaii International Conference on Computing Systems and will be compiled into a journal submission. This chapter profiles service resource demands on-line. The On-Line Service Placement Problem is defined. Using the on-line service profiles, caching strategies for the On-Line Service Placement Problem are theoretically predicted and empirically demonstrated.

Chapter 6 employs control theory to improve overall performance by employing a feedback control based on performance error. Theoretical and empirical results demonstrate a significant improvement on performance using the control versus not using the control. Additionally, the dynamic vector packing problem is examined. This work is being compiled into a future journal article. The document finishes with the conclusion and bibliography.

II. Context

Introduction

The brief overview in this chapter provides background to the nature of problems presented in the dissertation. Many problems in computer science have static, dynamic, and on-line versions of the same problem. The differences between the three are presented in this chapter with emphasis on the on-line version. After which a four level abstraction of service management is presented. Each layer abstracts information for the layer above. This abstraction allows for the mechanisms and attributes at each layer to be independent of the layer above and or below.

Static, Dynamic, On-Line

Static problems are the simplest form of computer science problems. Before the algorithm or heuristic used to solve the problem executes, all information is known and remains unchanged until the algorithm is complete. The classic knapsack problem has a sack of finite non-negative weight capacity, *W*, and a finite set of items, *X*, each with a non-negative weight w_i and a non-negative profit b_i. The goal is to fill the sack with items, *S,* to maximize profit and not over fill the sack.

$$i \in S \subseteq X, \max \sum b_i \text{ subject to } \sum w_i \leq W$$

In the static form, the number of objects remains fixed, the size of the sack remains fixed, and the weight and profit of the items remains unchanged. In another static example traveling sales person problem, the problem is to visit all cities one at a time while traveling the least distance. Each city is a node in the graph with each edge representing the distance. The graph representing the cities remains static and unchanged, e.g. the cities do not move. While static is the simplest form of a problem, this simplicity does not detract from the implications of the NP-Hard or NP-Complete nature of the problem.

Dynamic Problems are static problems over time. For each time step certain aspects of the problem can change. These changes are, however, known by the algorithm. For example, the knapsack problem could be expanded such that items arrive over time and additionally have an expiring profit. The algorithm must consider the optimal solution over time considering implications of profit decay. For example, in Manohar et al [17], a dynamic model creates a multi-period virtual topology in routed optical networks. A graph represents configurations as vertices and reconfiguration costs as edges. For each time interval, a specified number of configurations are greedily constructed, each with a fitness score and each assigned to a vertex of the graph. Each interval's configurations are fully connected to the next time interval's configurations. The edges specify the cost of change from one configuration to the next configuration. The solution to the problem is the path from the first interval configuration vertex to the final interval configuration vertex minimizing the fitness and the cost.

Online problems differ from the static and dynamic problems because the online algorithm does not have complete knowledge of the set of inputs. Generally, this is due to inputs arriving in the future. Albers [18] describes the online algorithm as receiving a sequence of inputs. These inputs must be completely processed as they are received. The future inputs are not necessarily related to the inputs previously received. An example of an online problem is the memory paging problem. Solutions to this problem include algorithms such as First In First Out (FIFO) and the Least Recently Used (LRU).

The online algorithm performance analysis compares the online algorithm's performance to an offline algorithm's performance. The off-line algorithm is aware of the entire sequence of inputs. This method is called *competitive analysis*. The cost of processing a sequence of inputs determines the performance measure. This measure is calculated for the online and the offline. The ratio of these two values determines the quality of the online algorithm.

Motwani and Prabhakar [19] define an algorithm *A* as *C-competitive* if there exists a constant *b* such that on every sequence $\rho_1, \rho_2, ..., \rho_N$

$$C_A = f_A\left(\rho_1, \rho_2, ..., \rho_N\right) - C \times f_0\left(\rho_1, \rho_2, ..., \rho_N\right) \leq b \tag{1}$$

where the constant *b* must be independent of *N*.

The sequence of inputs can be strictly random. This case is called the oblivious adversary. The *oblivious competitiveness coefficient* of *a* is then defined in [19] as

$$C_A^{obl} = \mathrm{E}\left[f_A\left(\rho_1, \rho_2, ..., \rho_N\right)\right] - C \times f_0\left(\rho_1, \rho_2, ..., \rho_N\right) \leq b \tag{2}$$

The *k*-server problem is a problem similar to the On-Line Service Placement Problem presented in this work. There are *k* mobile servers each residing at a point in a multi-dimensional space. A sequence of points in the same multi-dimensional space arrives as inputs, one at a time. The algorithm must select one of the *k* servers to "service" the requested point by moving the server from its current location to the requested point. The cost of the move is defined as the Euclidean distance from the server's current position to the requested point. The goal is to minimize the cost.

The greedy algorithm *in general* is extremely efficient for the *k*-server problem but is easily tricked, such as the example from Bartal [20]. This demonstrates the greedy algorithm is not competitive. Grove [21] has shown the Harmonic algorithm competitive in addressing the *k*-server problem with an upper bound of the competitive coefficient of $\left(5k2^k\right)/4$ for all *k*. The Harmonic algorithm calculates probabilities based on the inverse of the server's distance. From Raghavan and Snir [22], the probability of selecting the j^{th} server is

$$\mathrm{P}\left(x=j\right)=\frac{\frac{1}{d_j}}{\sum_{i=1}^{k}\frac{1}{d_i}} \tag{3}$$

The On-Line Service Placement Problem presented in this document is notionally similar to the *k*-server problem. Both problems use a coordinate system to represent a space of qualities or resources. The *k*-server problem has *k* servers available to handle individual requests, one at a time. In the On-Line Service Placement Problem, the nodes (servers) provide resources represented in a multi-dimensional space. Services placed on nodes consume resources similarly represented in the multi-dimensional space. The vector sum of the services on each node creates a point in the space. Minimizing the distance between these points to their respective node's point optimizes the placement. The *k*-server problem is only concerned with minimizing the movement cost. The service placement problem wants to minimize a cost (changes), have no under-provisioning, and minimize over-provisioning. Furthermore, services persist over time after arrival.

Metrical Task Systems were introduced in Borodin et al [23]. Albers [18] describes them as follows. Metrical Task Systems represent a framework for modeling a large class of on-line problems. First, a metric space is a pair (*S*, *d*) where *S* is a set of *n* states and where a distance function $d : S \times S \to \mathbb{R}_0^+$, where $d(i, j) \geq 0$ is the cost of changing from state *i* to *j*. Each task *T* is a vector $T = \langle T(1), T(2), ..., T(n) \rangle$, where $T(i) \in \mathbb{R}_0^+ \cup \{\infty\}$ is the cost of processing the task in state *s*(*i*). Given a sequence of tasks, an algorithm determines the schedule of states $s(1), s(2), ..., s(m)$. The cost of serving the task sequence is the sum of the processing and transition costs.

$$\sum_{i=1}^{m} d\left(s(i-1), s(i)\right) + \sum_{i=1}^{m} T^i\left(s(i)\right) \tag{4}$$

The On-Line Service Placement Problem is a metric task system. Let the matrix *S* represent the set of services where each row is a service and each column is the demand for a particular resource. Each T^i is the same as the service matrix S_i. Let *C* be the adjacency matrix mapping the assignment of services to nodes. The configuration matrix C_i is the state *s*(*i*). The function *d* is the 1-norm of the difference matrix $\left\| C_i - C_{i+1} \right\|_1$.

Albers [18] restates four theorems from the works of Borodin [23], Barta [24], and Fiat [26]. (1) There exists a deterministic online algorithm that is (2*n*-1)-competitive for any metrical task system with *n* states [23]. (2) Any deterministic online algorithm for the metrical task system problem has a competitive ratio of at least 2*n*-1, where *n* is the number of task system states [23]. (3) There exists a randomized online algorithm that is $\mathrm{O}\left(\log^2 n / \log^2 \log n\right)$–competitive against any oblivious adversary for any metrical task system with *n* states [26]. (4) Any randomized online algorithms for the metrical task systems problem has a competitive ratio of at least $\Omega\left(\log n / \log^2 \log n\right)$ against oblivious adversaries, where n is the number of task system states [24]. The first two theorems, given the possible number of states of C, imply the deterministic algorithm is not competitive. The last two theorems imply a random algorithm has acceptable performance.

Representation Layer

The purpose of the representation layer commoditizes the executable nature of the services.. This commodity by which the logic and information of the service is manifested in a binary representation. This representation is either executable or interpreted code. This code ranges from machine executable binary code to a high level scripting such as Haskell. Along with the executable code, the required data includes any additional information (beyond the logic). This data can take the form of text files, symbol tables, raw images, or SQL databases. These two pieces of binary data constitute the service. The Representation layer is analogous to Layer 0 in the OSI model.

For example, if a service is to provide the average temperature for a specified date range and location, the data can be acquired and a web service can be written to delivery the requested information. One possibility is the data is a text file and the service a binary executable that can be run from a command line or as a Common Gateway Interface application in a web server. Alternatively the data could be stored in a SQL database file and the service written in Perl. The point is the 1’s and 0’s may change or may be moved without changing the semantics of the services. The representation layer abstracts these considerations from the above layers more concerned with higher level issues.

Alternatively consider the representation as an bootable ISO image or a pre-loaded virtual machine file. These files are prepared for execution in the appropriate virtual machine engine such as KVM, Xen, or VMWare. In this case the commodity is includes the operating system, web server, and the service(s). Regardless the representation the services can be moved, duplicated, deleted, monitored, and executed.

Consider the human management of services. At this level there is no higher level consideration than the single individual service. Managing systems at this level is ad-hoc at best. System administrators move or change services in response to trouble or new requirements that springs up or when it appears to be a good idea, i.e. using fewer servers. Using Carnegie Mellon’s Capability Maturity Model Integration (CMMI)[26] as an analogy, the Representation layer is similar to managing at the CMMI Level One in which decisions are made without metrics, measures, or procedures.

Consider the Representation layer providing an standard interface so that managing and handling the underlying binary of a service like how the payload of an IP packet is handled. The above layer, described next, is provided standardized information from this layer regardless of how it is manifested. As described next, this layer has or creates measures such as size, response times, cost of movement, etc. This layer is indifferent to any of these values as it is only concerned with the commodity itself.

Quantification Layer

The quantification layer builds upon and abstracts the representation layer. The quantification is a numeric representation of the lower layer. This numeric representation creates an ordering of equivalence classes across nodes, services, and the assignments of the services to nodes. These values are assigned to or captured from the environment. The values can be static or dynamic. These measures can be a real, integer, Boolean, enumerated, single valued, multi-valued, etc. In some cases a probability distribution can be used to quantify behavior.

Static values are assigned or derived but remain constant such as the size of the executable code of a service. Another example is a ceiling on the maximum or minimum amount of memory required by a service. A probability distribution could provide a form of a static measure, assuming the distribution does not change. Static values are simpler to measure or assign than dynamic values which require ongoing measurements or assignments. These static measures of the services are determined directly from the code provided or are determined by running the service "offline" and generating values that approximate online behavior. For example, workload varied overtime in an offline environment creates a min, max, and mean for memory usage. A complete probability distribution can be created as well for CPU usage [27].

While easy and simple to collect, the trouble with static measures is their inability to model changes over time in a live environment. They are sufficient if the environment is constant or will reach an eventual steady state. Burstiness is difficult to discuss in the context of static distributions.

Dynamic measurements are collected from sensors placed into the system. These sensors capture data on a continuous or discrete time scale. These values of a service come from built in operating system monitoring tools which report CPU, memory, and network utilization by processes. The information could be extracted from log files tracking web server hits or database transactions. Nearly all modern hardware and operating systems support management interfaces like Simple Network Management Protocol (SNMP) allowing access to performance details.

These measures are also derived for the node describing both the capacities of the node resources and the usage of those resources. The services are measured by the resources each is consuming or has consumed. Additionally, the service has internal measures of performance, i.e. how many transactions have been processed. Issues of interest at this layer primarily involve (1) how the sensor manifests the information, (2) how to aggregate the measures, (3) how to transmit the measures, (4) where to store the measures (centrally, regionally, individually), (5) how long to store the information, and (6) how the data can be fused at a low level for better storage, retrieval, and eventual interpretation.

In summary, the quantification layer is similar to the CMMI Managed and Defined levels in which procedures and products are documented, counted, and measured; although, no value judgments of processes are made. Improving processes happens at the next level. This is strictly the matter of fact processes used today. Similarly, the IP and TCP headers provide measurements, purpose, and destination of the otherwise meaningless payload. The quantification layer provides the next layer (qualification) with the information necessary to make judgments and decisions. The following are a few items from the literature about quantifying the supply and demand of resources.

The Job Submission Description Language (JSDL) Specification [28] is an extensive XML specification for submitting jobs to Grid environments. These values are maximum and minimum capacities required to service the job, which is generally assumed to be a batch processing job such as processing through large amounts of data. The available attributes are extensive, from CPU and operating system type to pipe size and stack size limits.

Jenkins and Rice [29] discuss the general properties of diverse resources drawn from a wide spectrum of domains where simulation engines execute. They define a Resource Typology based on four main areas of Existential, Availability, Utility, and Implementation. The Existential category covers properties related to a resource's independent existence, including its identity and whether using the resource fundamentally changes it. The Availability category bears upon variations in when and to whom a resource provides service. The Utility category deals with what a resource does, how much of a resource exists, how well the resource performs, and what cognitive properties the resource exhibits. Implementation characteristics involve differences in the way two resources provide the same service. Table 1 provides a detailed list for each category.

Table 1. Resource Typology

Existential	**Availability**	**Utility**	**Implementation**
Identity	Status	Competencies	Adaptability
Origin	Location	Size	Activity
Living	Schedule	Performance	Interactivity
Consumption	Delivery mode	Deliverability	Autonomy
Make-up	Failure mode	Reliability	Coupling
Traits	Selectivity	Effectiveness	Isolation
	Exclusivity	Efficiency	Discoverability
		Cost	Composition
		Quality	Centralization
		Cognition	Mobility
			Forgetfulness
			Preemptibility
			Standardization
			Risk
			Policy

Stewart and Shen [30] empirically profile two multiple tier applications. They express the CPU, disk, and memory of the web server and database server of each application component as a linear function of the workload and overhead. Based on their model they devise a method to predict throughput and response time prediction. These models are compared in varying environments to determine impact to heterogeneous systems, cluster sizes, replication strategies, and request mixes.

Qualification Layer

The qualification layer assumes information collected at lower levels is accurate and sufficient to make decisions based on the current environment and possible, considered environments. While the policy objectives are determined in the organization layer (next layer above), the qualification layer provides the mathematical and logical mechanisms to implement those objectives. In general the objectives minimize (or maximize) some function or functions. Given a set of inputs, the functions, however simple or complicated, provide an output qualifying the set of inputs with a partial ordering. This ordering creates equivalence classes. Using the equivalence classes and their ordering aggregates and segregates items, thus allowing for policy to be applied.

The inputs are from (1) raw information in the quantification layer below and (2) parameters established in the above layers. These parameters include dollar costs, energy costs, or other arbitrarily assigned weights expressing particular preferences such as preferred customers. These inputs influence the quality of the existing configuration. The assigned quality of the existing configurations is compared to alternative configurations to determine the best available configurations.

In the literature, the evaluation of these objective functions takes many forms. The general class of problems is combinatorical multi-constraint, multi-dimensional optimization problems. For example, game theory considers the Nash Equilibriums, the core, and the kernel. In graph theory, these functions are expressed in minimum spanning trees, bipartite matchings, graph coloring, minimum cost shortest path, and many others.

Integer, linear, and non-linear optimization techniques provide solutions to these problems. Biological inspired techniques such as swarm and ant algorithms also provide solutions. Partially Observable Markov Decision Process and other Operational Research methods can determine optimal or near-optimal solutions. The choice of methods depends on at least five factors: (1) how optimal of a solution is required, (2) how much time the technique requires to arrive at a solution (or multiple solutions), (3) what is the certainty of the inputs, (4) what is the scale of the problem, and (5) can the problem be subdivided or decentralized.

At this layer consideration must be given to the churn of the environment. The quality of the current configuration and potential alternatives must be controlled to avoid thrashing. System partitioning, spikes in demand, and dips in resource availability must be handled properly.

In summary, the qualification layer is similar to the CMMI Defined and Qualitatively Managed levels where the tools for analysis, validation, and verification are defined. In terms of the OSI model, the higher level functions such as routing mechanisms, session management, and presentation abstractions enable mission objectives to be met without concern for lower level implementations.

The POMDP process [35][36][37] is based on the Markov Decision Process where the underlying states and probabilities are not directly available but are ascertained from observations. These observations allow a working model to be created and allow best effort decisions to be made. It is often implemented in multi-agent artificial intelligence scenarios. In Castanon [38], they utilize POMDP to consider a class of unreliable resource allocation problems where resources assigned may fail to complete a task and the outcomes of past resource allocations are observed before new resource allocations are selected. They create an approximation of the model and optimize it with good results. In [37], they derive action policies and communication policies that optimize a global value function using an analytical model to evaluate the trade-off between the cost of communication and the value of the information received.

The classic knapsack problem is given a set of items with a weight and value (profit) find the combination of items that profit and do not over fill a sack that can hold a fixed amount of weight. The classic knapsack problem is NP-hard [39]. Algorithms can solve this in pseudo-polynomial time [39]. The literature and textbooks mention using linear programming with relaxation, simulated annealing, branch-and-bound techniques, and dynamic programming. Solving the multi-objective, multi-dimensional, 0,1 knapsack is NP-Hard [39]. Akbar et al [39] present a heuristic based on convex hulls leading to a quick solution that is 88%-98% optimal. It is used for the admission controllers for multi-media systems that need quick answers. Similar to their motivation, the end goal is an on-line algorithm that can provide an acceptable solution in a short amount of time.

In Bartlett [40], they describe the Temporal Knapsack Problem based on a multiple CPU task scheduling. They describe it as a generalized form of knapsack and a specialized form of the multi-dimensional knapsack. They compare and contrast various algorithms to address the matter.

Lau [32] discusses the behavior of the multi-dimensional knapsack problem in various scenarios with demand (low/high), resources (unlimited/limited), and profit (fixed/variable). Pisinger [42] addresses the multiple objective knapsack problems using a form of dynamic programming. Bertsimas [43] uses what he calls approximate dynamic programming to tackle the multi-dimensional knapsack problem (MMKP). Han [44] states in their introduction, "Most academic efforts related with MMKP have been put on finding heuristic algorithms due to the NP-hard nature of the problem."

In other autonomic service-oriented architecture works [45][46][9], the focus is based on maximizing profits while maintaining multiple tier service levels. Almeida et al. in [45] breaks the resource allocation problem into a short term arrangement problem and a long term allocation problem. They utilize queuing and optimization techniques in their model and performance measurements. Ardagna et al in [46] similarly model and measure queuing and optimization techniques to assign virtual machines to CPUs in a physical server. A self-adaptive capacity management framework described in [9] uses queuing and optimization in a multi-tier virtualized environment demonstrating significant profit gains relative to a static model.

Organization Layer

The Organization Layer provides a place for autonomous and human directed policy to reside. System directives and policies establish priorities which implement the overall quality of the system. Considerations here emphasize profits, costs, and security.

Service level agreements between customers and providers are established here and communicated downward into the framework. These agreements are then subject to the feedback from the qualitative layer below. Profits are measured in monetary gains, available additional capacity or simple catastrophe avoidance. Costs are contained in terms of minimal movement, energy reductions or the avoidance of prescribed penalties. Security is measured in overall availability, confidentiality and integrity of the system and the customers' applications. This layer, for example, provides the priorities (derived from profit, costs, and security) if the system encounters a difficult environment and must determine which services suffer and which services are granted preferential treatment.

The most challenging aspect of this layer is translating the desired policies into parameters for delivery to the qualitative layer. These numeric values describe the intention of the system. Part of the challenge is to avoid contradictory and competitive values. The other part is how all polices are modeled in terms of quantities. Under certain conditions this may be in the form of a feedback loop. As information is returned from the qualitative layer, adjustments to parameters are autonomically made to keep the system in balance. Alternatively, manual intervention could be required in adjusting parameters.

Summary

This chapter outlines the On-Line Service Placement Problem as an on-line metrical task system knapsack problem. Related research was presented showing the similarities and differences. The four layer framework subdivides the problem into abstract levels where facets of the problem can be isolated from each other. This allows for the mechanics at each level to change independent of the layers above and below it.

The rest of this work develops methods and parameters for implementing policy. Performance policy is specifically addressed. Performance policy specifies the extent service performance is preferred to service inclusion. Performance policy specifies the performance threshold that triggers action to be taken. To implement these policies, values representing the nodes and services are developed in the quantification layer. Methods for finding and implementing configurations are developed in the qualification layer. For these methods, parameters are available to translate the policy into math.

III. Testbed

Introduction

This chapter discusses the testbed, called the Cloud Chamber, developed and used throughout this work for empirical investigations. It provides an environment for the placement of services on node resources. Testing clients execute services based on pre-defined loads. These loads produce performance data for both the services and the nodes. Each of the experiments uses the Cloud Chamber in different ways. Chapter 4 uses the testbed with static services modeled a priori and pre-calculated configurations. Chapter 5 uses on-line service profiling and on-line configuration generation. Chapter 6 incorporates performance feedback loops to influence the on-line configuration generation. This chapter describes in detail the foundation of the Cloud Chamber and details about how each experiment utilized its features.

Physical Architecture

The architecture of the testbed is comprised of commercial off the shelf (COTS) hardware and software as well as custom applications. The testbed’s physical hardware includes three servers and a workstation. The primary server, HP Proliant ML330G6 with dual Intel Xeon E5620 processors and 36GB of memory, runs VMWare ESXi 4.0. This server houses the virtual web servers and is depicted in the upper left of Figure 3. The secondary server runs VMWare ESXi 4.0 with an Intel Q8400 Quad Core and 4GB. This server, depicted in the upper right of Figure 3, houses the virtual machines with services that consume the web services (as ah http client). The third server with an Intel Pentium

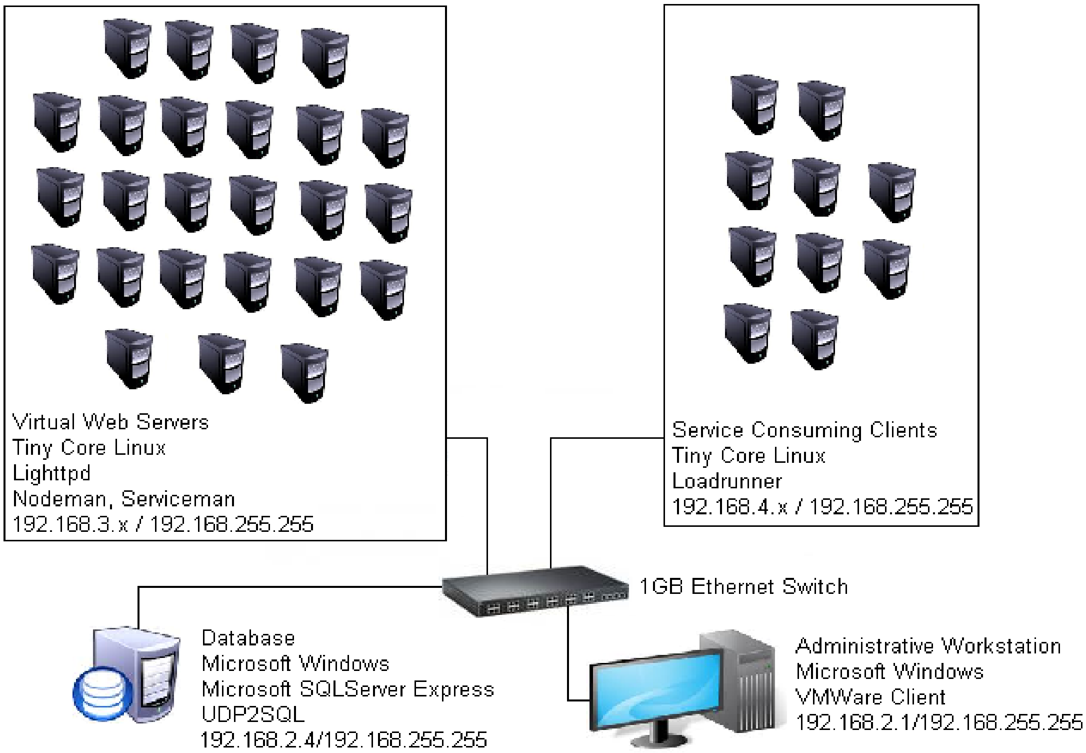


Figure 3. Cloud Chamber Architecture

Dual Core 3.0 GHz and 3GB houses a SQL database engine for data collection and storage and a web server with administrative pages for controlling the overall experiment. This server is depicted in Figure 3 on the bottom left. Additionally, a workstation is used for data aggregation and analysis using tools such as MATLAB. This workstation is depicted in Figure 3 on the bottom right.

The primary VMWare ESXi 4.0 [47] server houses twenty-five virtual web servers (nodes with resources). Each is an individual 64MB VMWare image. These virtual machines have TinyCore Linux 3.0 [48] installed as the operating system. TinyCore is a minimal Linux installation with only essential services included. It can have as small as a 10MB installation footprint and use as low as 64MB of memory. Although TinyCore can boot directly from a CD or USB stick, the default persistent

TinyCore installation is the basis for the machine. See TinyCore documentation for further details. On each virtual web server, DHCP is disabled with a hard coded IP address in the 192.168.3.x range and net mask of 192.168.255.255. In addition to the minimal core of Linux, a small web server, *lighttpd*, is installed. In VMWare ESXi, each virtual machine can have limits on its virtual hardware resources such as CPU speed, memory capacity, hard drive capacity, and available network bandwidth. By varying these resource limitations across the virtual web servers, a heterogeneous set of compute resources is created. Each experiment in this dissertation uses a different set of resource limitations per machine. Each virtual web server also executes two custom applications: *serviceman* and *nodeman*. These applications and their interactions are summarized in Figure 4 and described below in more detail. These virtual web servers provide the compute resource for the services in the Cloud Chamber to be executed.

Load

The secondary VMWare ESXi server houses the service consuming clients. These virtual machine clients use TinyCore as above but only have a single custom application, *loadrunner*. The *loadrunner*, based on *http_load* [49] and similar to *httperf* [50], executes the requests on each service. This prescribed request load (described in detail below) for each service is the number of hits per second for each time step over the span of the experiment. Each *loadrunner* is capable of simultaneously handling the requests for multiple services. As determined in the initial testing and sandboxing, the *loadrunner* application is able to deliver up to a combined total of 500 http requests per second. In order to achieve (system wide) more than 500

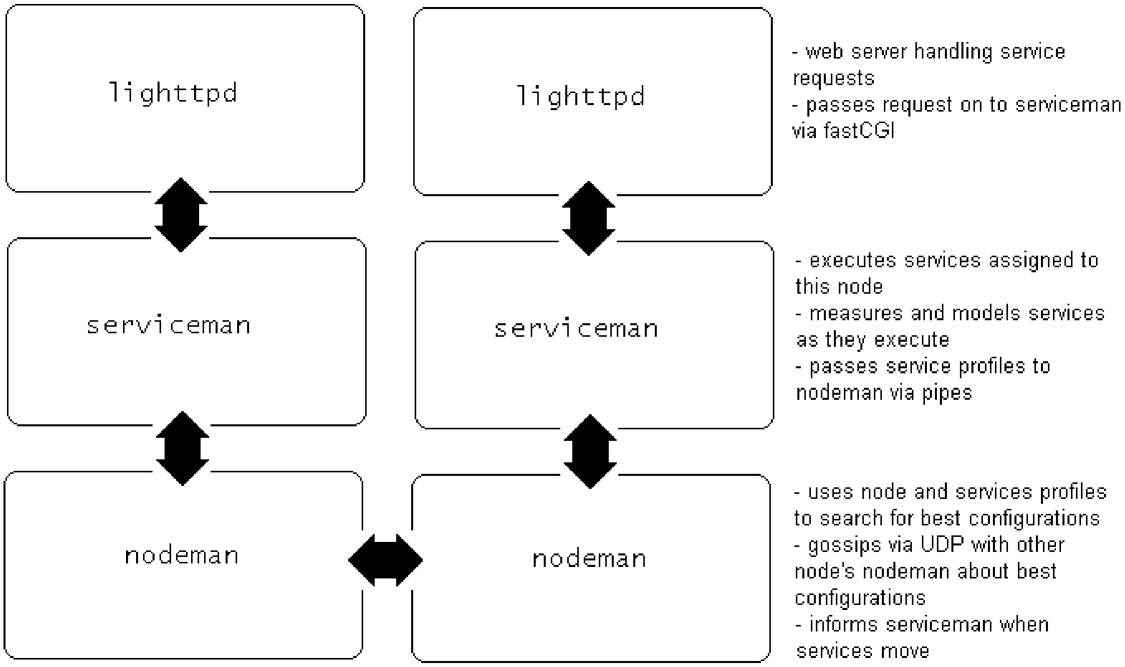

lighttpd
lighttpd
- web server handling service requests
- passes request on to serviceman via fastCGI
serviceman
serviceman
- executes services assigned to this node
- measures and models services as they execute
- passes service profiles to nodeman via pipes
nodeman
nodeman
- uses node and services profiles to search for best configurations
- gossips via UDP with other node's nodeman about best configurations
- informs serviceman when services move

Figure 4. Inside the TinyCore Node

http requests per second, multiple instances of *loadrunner* are deployed to each virtual machine. Each virtual machine executes 10 instances of the *loadrunner* application. Depending on the traffic load, each instance can handle 10 services. With 10 virtual machines each with 10 *loadrunner* instances, up to 1000 services can be consumed simultaneously. In the initial testing and sandboxing, a system wide total of up to 2000 hits per were executed while maintaining system wide stability. Note: this aggregated performance varies depending on service payload size and total number of services.

The *loadrunners* request http and expect http responses as defined by the RFC 2616 [51]. Successful executions of the service return an http status code of 200. If a service is moved from one web server to another, the next request by the *loadrunner* in charge of the moved service receives a *301 Moved Permanently* http status code with the new IP address in the content of the response, per the RFC. The *loadrunner* updates its information, and all new requests are directed to the new IP.

Each time step, the *loadrunners* query the database server for the prescribed hits per second for each service. Each *loadrunner* then adjusts its internal timer to execute this prescribed number of hits per second. The aggregate set of 10 *loadrunners* can handle 500 hits per second. Above this limit, the physical CPU saturates. This saturation skews the collected data such as response time. In particular, the virtual switch maintained by VMWare can not provide the prescribed throughput. As requests are returned, *loadrunner* aggregates performance information such as time to connect, response time, bytes returned, etc. Every few time steps, the *loadrunners* send this information to the database server for collection.

Services

Each service is executed as a fast common gateway interface (FastCGI) executable written in C. FastCGI [52][53][54] is a protocol defining the method to pass a web request from a web server such as Apache, IIS, and *lighttpd* to an application server such as Perl, PHP and to pass the response from the application server back to the web server. In the early days of http web servers, Common Gateway Interface (CGI) applications were executable code invoked by a web browser using the same http request for html pages. The web server executed the requested standalone program which returned its standard out to the requesting browser. As demand increased, the operating system overhead of launching a new process for each request became cumbersome. This problem led to the FastCGI protocol. A FastCGI application is launched when the web server boots. Each request to the web server is passed to the FastCGI application via a socket. The output returns over the same socket to the web server and subsequently returns to the requestor. In the Cloud Chamber, the custom FastCGI application, *serviceman*, on each virtual web server emulates a web service by taking its request, consuming a prescribed amount of system resources for that service, and returning its payload.

Currently, each service consumes CPU cycles, memory, and bandwidth based on scalar values assigned to each service. For example, a service described respectively by <100, 2000, 40> executes a *for* loop 100 microseconds, uses 2000KB of memory, and returns 40KB of payload over the network. These values define the service and represent the service's code behavior. Additional resource consumption types such as reads and writes to permanent storage could easily be added. These values are stored in a service file like service1.svc. An http request to the web server arrives in the form http://10.10.10.10/service1.svc. The service file is passed from the web server to *serviceman* using the FastCGI protocol. *Serviceman* proceeds to consume the CPU and memory and returns a payload of prescribed size to the web server. The web server completes the transaction by returning the payload to the requester thus consuming the network bandwidth. Alternatively, the resource values in the file can be overridden by passing the new values on the request, for example, http://10.10.10.10/service1.svc?cpu=300.

The above numbers were simplified for illustration. In order to accommodate services of various shapes and sizes, the Cloud Chamber currently supports three distributions from which the consumption values are derived. The values are stored in the service (.svc) file as parameters to distribution functions. The first possible distribution is the fixed distribution in which the value is the same for every request for the service. The second is the uniform random distribution in which a lower bound and upper bound prescribe a range over which the value will be uniformly derived. The third is the standard distribution in which a mean and standard deviation determine the random values. Other options such as Poisson, Exponential, etc. can easily be added but were not available as of this writing. As mentioned above, if the tester wishes to override these values, the alternative values are passed to *serviceman* via parameters in the URL.

In developing this consumption emulator, *serviceman*, a handful of notable observations were encountered, although not novel to the literature. Primarily, these derived from compiler and operating system optimizations. For memory, a simple *malloc()* call is insufficient to consume memory. First if the requested memory is never accessed, the operating system (in this case a recent flavor of Linux, TinyCore) does not necessarily completely allocate the memory. Second, the service execution times, e.g. time between *malloc(),* memory accesses, and *free(),* were sufficiently small that utilities for system monitoring, like *top*, did not sample the system fast enough to see substantial memory utilization. The current work around is that *serviceman,* upon the first invocation of a service, allocates memory which persists between invocations of the service. This can be thought of as the service’s stateful memory requirements. The service’s stateless memory requirements are also allocated and deallocated during each invocation.

CPU cycles are consumed using a *for* loop. The *clock_gettime()* function provides the number of nanoseconds for which a process or thread has executed on the CPU so far. This function is called at the beginning of the consumption, before the memory has been allocated and accessed and data has been sent to the network. This allows for the capturing of the CPU time consumed by memory consumption and data sent to the network. After these two tasks, the *for* loop is executed a specified amount of time in microseconds emulating additional CPU consumption. Compiler optimization issues arose when attempting to burn CPU cycles. In one example, writing the same value to the same place multiple times did not take as long as expected. In order to avoid any IO blocking and other opportunities for the service to be preempted, the *for* loop is a loop performing only an increment of a counter.

Each service's resource consumption is compiled by *serviceman*. The CPU time, bytes of output, memory consumption, and request count are continuously accumulated. Each second, a histogram data structure for each resource's consumption is updated with the accumulated resource consumption data. The histogram tracks resource consumption per request. From the histogram, a linear function is regressively derived using Box-Muller [55]. This linear function given any level of requests per second will output the predicted resource consumption for the service. If the near future traffic levels are known or can be calculated, the anticipated resource consumption of the service can be calculated. The mean hits per second for the last 60 seconds is used in many of the experiments in the dissertation. In others the mean of the last 2 seconds is used. (The choice of method for predicting future traffic loads is a policy decision and is out the scope of this dissertation.) Using the predicted value, the resource profile for each service is calculated and handed via named pipes to the custom application *nodeman* for dissemination.

It should be noted for Chapter 3 the above dynamic profiling was not used. Services were profiled off-line before the experiment to create a static profile. Furthermore, configurations were generated off-line using MATLAB and fed over time into the system. This was due to the need to frame the static problem as well as because the testbed was still under development when the Chapter 3 experiments were executed.

The custom application *nodeman* executes on each web server (node) and works in conjunction with *serviceman*. The first function of *nodeman* is to organize the web servers into a topology for communication. The second function is to determine the placement of the services on the nodes. Using UDP datagrams the *nodeman* process on each node communicates with each other. The nodes' conversation establishes a directional ring topology where each node has a neighbor. Each node gathers node profile and service profile information about itself. These profiles are passed to its neighbor and, for speed and redundancy, to a random node across the ring. Any information the node receives is recorded internally and passed along as above. After a specified amount of time, the nodes settle on the intersection of all profile information. The agreed upon profiles of the nodes and the services is then used to search for an appropriate placement. Each node uses the greedy heuristic (described later in this chapter) and quality function (described later in the next chapter) to search for a better placement of services on nodes. If a node finds a better configuration, the configuration is passed around to other nodes as described above. If a node receives a better configuration, the configuration is noted internally and passed along as above. After a specified amount of time, the nodes settle on a new configuration. After which the services are placed as prescribed by the new configuration. The process is then repeated by nodes conversing about the new profiles including possible new nodes and services. This process is discussed in more detail in subsequent chapters.

From all of the outlined processes occurring in the Cloud Chamber, performance data regarding nodes, web servers, and services is collected. *Serviceman* reports data every second about the node’s performance including CPU utilization, memory consumption and bandwidth. This node operating system information is logged to a table in the database named *noderaw*. *Serviceman* also reports every second the number of responses handled and other data regarding the web server in a table named *svcraw*. *Nodeman* reports to a table *nodemanraw* regarding the events of the organization of nodes and services. *Loadrunner* reports every second about its requests such as response times, open connections, http codes, and other supporting data. This request data is logged to the *loadraw* table.

This aforementioned data is sent to the SQL Server via a UDP datagram. The custom application *udp2sql* executing on the SQL Server receives the UDP datagram. The application extracts the payload, an SQL INSERT or UPDATE statement, locates any placeholders such as “timestamp”, and replaces them with the appropriate value. The timestamp place holder allows for the events to be totally ordered per the time they are processed by *udp2sql*.

The data collection was spread over several time steps to avoid saturating the database server. Prior to this thinning, UDP datagrams were being lost because *udp2sql* was unable to process them as fast as they arrived. Initially, *serviceman* sent one UDP datagram per service and one for each web server. *Loadrunner* sent one per service. *Nodeman* periodically sent bursts of a few dozen. These totaled approximately 250 datagrams per second. The *loadrunners* were turned down to send once every ten seconds and set such that not all 100 arrived at the same time by randomly off-setting the start time.

The database has other supporting tables. These include tables for *services*, *nodes*, *loadrunners*, etc. The records in the *loadrunners_services* table assign a service to a *loadrunner*. The table *prescribed_load* provides the traffic load for the entire experiment by assigning the number of hits to the specified time step for the service. Every second, as the experiment executes, the records for that second are copied from *prescribed_load* to the *loadrunners_load* table. The *loadrunners* each query this table twice per second adjusting the rate each service is consumed. Figure 5 is the database entity relationship diagram. Tables 2 through Table 5 are sample records from selected database tables.

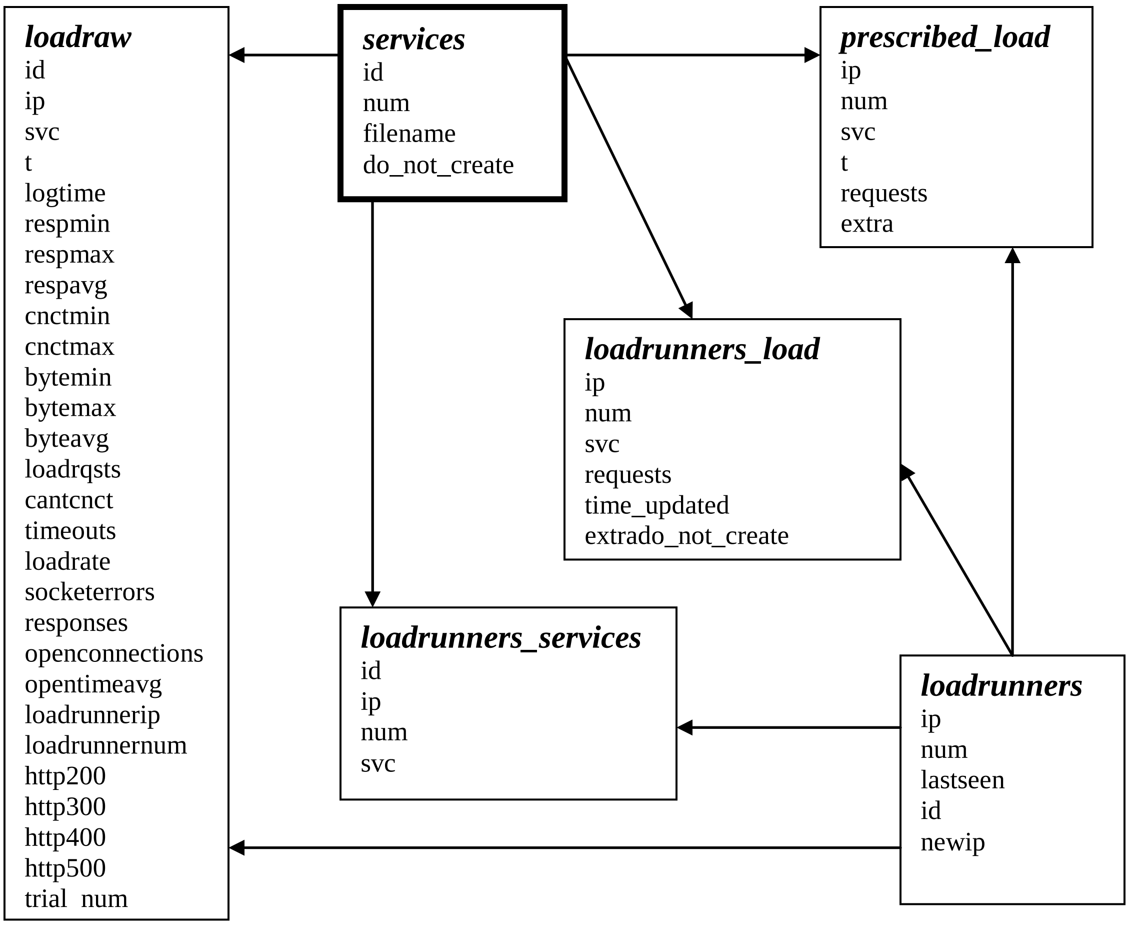


Figure 5. Database Entity Relationship Diagram (Part 1)

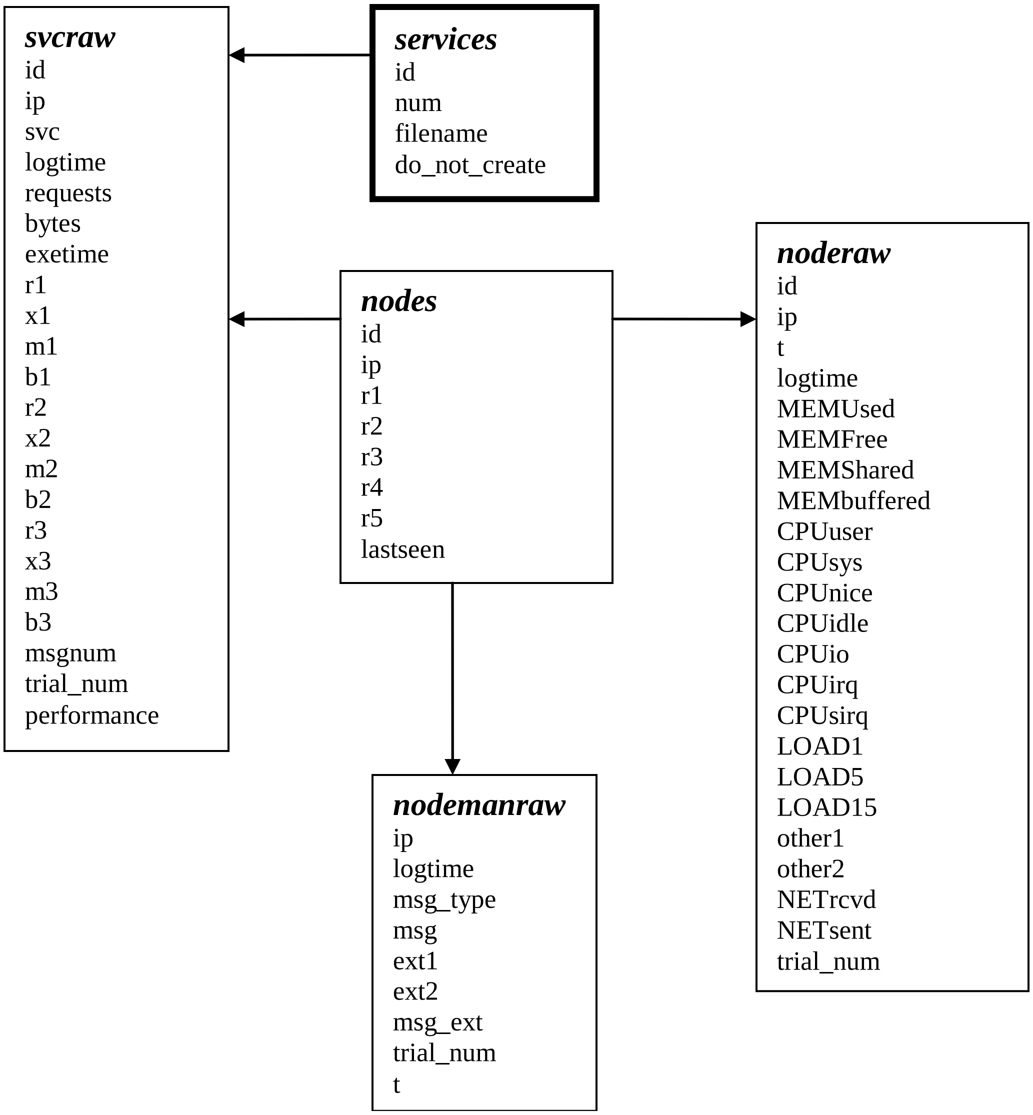


Figure 5. Database Entity Relationship Diagram (Part 2)

=

Table 2. Nodes SQL Database Table with IP and Profile.

id	ip	r1	r2	r3	r4	r5	lastseen
4	192.168.2.11	1	1	1	NULL	NULL	20:25.7
5	192.168.2.12	1	0.25	0.01	NULL	NULL	20:25.9
6	192.168.2.13	0.4	1	0.01	NULL	NULL	09:47.7
7	192.168.2.14	0.4	0.25	1	NULL	NULL	09:47.5
8	192.168.2.15	0.6	0.5	0.1	NULL	NULL	20:25.7
9	192.168.2.16	0.6	0.5	0.1	NULL	NULL	20:25.4
10	192.168.2.17	0.4	0.25	0.01	NULL	NULL	20:25.5
11	192.168.2.18	0.4	0.25	0.01	NULL	NULL	09:47.4
12	192.168.2.19	0.6	0.25	0.01	NULL	NULL	09:47.3
13	192.168.2.20	0.6	0.25	0.01	NULL	NULL	20:25.4
14	192.168.2.21	0.8	1	1	NULL	NULL	20:25.9
15	192.168.2.22	0.8	0.25	0.01	NULL	NULL	09:47.2
16	192.168.2.23	0.4	1	0.01	NULL	NULL	20:25.5
17	192.168.2.24	0.4	0.25	1	NULL	NULL	20:25.3
18	192.168.2.25	0.6	0.5	0.1	NULL	NULL	20:25.7
19	192.168.2.26	0.6	0.5	0.1	NULL	NULL	20:25.3
20	192.168.2.27	0.4	0.25	0.01	NULL	NULL	09:47.1
21	192.168.2.28	0.4	0.25	0.01	NULL	NULL	20:25.3
22	192.168.2.29	0.6	0.25	0.01	NULL	NULL	09:47.9
23	192.168.2.30	0.6	0.25	0.01	NULL	NULL	20:25.5
24	192.168.2.31	0.8	1	1	NULL	NULL	20:25.5
25	192.168.2.32	0.8	0.25	0.01	NULL	NULL	20:25.7
26	192.168.2.33	0.4	1	0.01	NULL	NULL	20:25.3
27	192.168.2.34	0.4	0.25	1	NULL	NULL	20:25.1
28	192.168.2.35	0.6	0.5	0.1	NULL	NULL	20:25.2

Table 3. Services SQL Database Tabel with ID and URL (25 of 100)

id	num	filename
1	1	foo.svc?service=0001&r=5,p0=2095,p1=1,p2=0&r=5,p0=1339,p1=1,p2=0&r=5,p0=15,p1=1,p2=0
2	2	foo.svc?service=0002&r=5,p0=1159,p1=1,p2=0&r=5,p0=1255,p1=1,p2=0&r=5,p0=18,p1=1,p2=0
3	3	foo.svc?service=0003&r=5,p0=2126,p1=1,p2=0&r=5,p0=1252,p1=1,p2=0&r=5,p0=7,p1=1,p2=0
4	4	foo.svc?service=0004&r=5,p0=1965,p1=1,p2=0&r=5,p0=1357,p1=1,p2=0&r=5,p0=6,p1=1,p2=0
5	5	foo.svc?service=0005&r=5,p0=2588,p1=1,p2=0&r=5,p0=1253,p1=1,p2=0&r=5,p0=15,p1=1,p2=0
6	6	foo.svc?service=0006&r=5,p0=1569,p1=1,p2=0&r=5,p0=1335,p1=1,p2=0&r=5,p0=32,p1=1,p2=0
7	7	foo.svc?service=0007&r=5,p0=1137,p1=1,p2=0&r=5,p0=1284,p1=1,p2=0&r=5,p0=6,p1=1,p2=0
8	8	foo.svc?service=0008&r=5,p0=1268,p1=1,p2=0&r=5,p0=1331,p1=1,p2=0&r=5,p0=6,p1=1,p2=0
9	9	foo.svc?service=0009&r=5,p0=1477,p1=1,p2=0&r=5,p0=1274,p1=1,p2=0&r=5,p0=8,p1=1,p2=0
10	10	foo.svc?service=0010&r=5,p0=1422,p1=1,p2=0&r=5,p0=1142,p1=1,p2=0&r=5,p0=29,p1=1,p2=0
11	11	foo.svc?service=0011&r=5,p0=2501,p1=1,p2=0&r=5,p0=1071,p1=1,p2=0&r=5,p0=24,p1=1,p2=0
12	12	foo.svc?service=0012&r=5,p0=2594,p1=1,p2=0&r=5,p0=1410,p1=1,p2=0&r=5,p0=16,p1=1,p2=0
13	13	foo.svc?service=0013&r=5,p0=1915,p1=1,p2=0&r=5,p0=1291,p1=1,p2=0&r=5,p0=10,p1=1,p2=0
14	14	foo.svc?service=0014&r=5,p0=1296,p1=1,p2=0&r=5,p0=1352,p1=1,p2=0&r=5,p0=27,p1=1,p2=0
15	15	foo.svc?service=0015&r=5,p0=1492,p1=1,p2=0&r=5,p0=1141,p1=1,p2=0&r=5,p0=12,p1=1,p2=0
16	16	foo.svc?service=0016&r=5,p0=2494,p1=1,p2=0&r=5,p0=1296,p1=1,p2=0&r=5,p0=6,p1=1,p2=0
17	17	foo.svc?service=0017&r=5,p0=1512,p1=1,p2=0&r=5,p0=1386,p1=1,p2=0&r=5,p0=6,p1=1,p2=0
18	18	foo.svc?service=0018&r=5,p0=1011,p1=1,p2=0&r=5,p0=1355,p1=1,p2=0&r=5,p0=7,p1=1,p2=0
19	19	foo.svc?service=0019&r=5,p0=1128,p1=1,p2=0&r=5,p0=1256,p1=1,p2=0&r=5,p0=24,p1=1,p2=0
20	20	foo.svc?service=0020&r=5,p0=1196,p1=1,p2=0&r=5,p0=1245,p1=1,p2=0&r=5,p0=26,p1=1,p2=0
21	21	foo.svc?service=0021&r=5,p0=1013,p1=1,p2=0&r=5,p0=1208,p1=1,p2=0&r=5,p0=28,p1=1,p2=0
22	22	foo.svc?service=0022&r=5,p0=2197,p1=1,p2=0&r=5,p0=1340,p1=1,p2=0&r=5,p0=15,p1=1,p2=0
23	23	foo.svc?service=0023&r=5,p0=1845,p1=1,p2=0&r=5,p0=1102,p1=1,p2=0&r=5,p0=12,p1=1,p2=0
24	24	foo.svc?service=0024&r=5,p0=1381,p1=1,p2=0&r=5,p0=1212,p1=1,p2=0&r=5,p0=29,p1=1,p2=0
25	25	foo.svc?service=0025&r=5,p0=2167,p1=1,p2=0&r=5,p0=1457,p1=1,p2=0&r=5,p0=5,p1=1,p2=0

Table 4. Loadrunners SQL Database Table with IP and Num (25 of 100 records).

ip	num	lastseen
192.168.3.100	4	26:07.0
192.168.3.100	5	26:07.0
192.168.3.100	10	26:07.0
192.168.3.100	1	26:07.0
192.168.3.101	2	26:06.0
192.168.3.101	4	26:07.0
192.168.3.101	8	26:07.0
192.168.3.102	10	26:07.0
192.168.3.102	2	26:07.0
192.168.3.103	6	26:07.0
192.168.3.104	1	26:07.0
192.168.3.105	1	26:07.0
192.168.3.107	2	26:07.0
192.168.3.108	4	26:07.0
192.168.3.108	10	26:06.0
192.168.3.106	7	26:07.0
192.168.3.101	9	26:06.0
192.168.3.102	9	26:07.0
192.168.3.103	4	26:07.0
192.168.3.103	8	26:07.0
192.168.3.103	10	26:07.0
192.168.3.104	6	26:07.0
192.168.3.104	5	26:07.0
192.168.3.105	8	26:07.0
192.168.3.106	2	26:07.0

Table 5. Prescribed Load SQL Database Table (25 of 241,000 records).

ip	num	svc	t	requests	extra
192.168.3.101	1	1	601	1	0
192.168.3.102	1	2	601	5	0
192.168.3.103	1	3	601	4	0
192.168.3.104	1	4	601	7	0
192.168.3.105	1	5	601	4	0
192.168.3.106	1	6	601	4	0
192.168.3.107	1	7	601	3	0
192.168.3.108	1	8	601	3	0
192.168.3.109	1	9	601	4	0
192.168.3.100	1	10	601	3	0
192.168.3.101	2	11	601	7	0
192.168.3.102	2	12	601	9	0
192.168.3.103	2	13	601	5	0
192.168.3.104	2	14	601	4	0
192.168.3.105	2	15	601	3	0
192.168.3.106	2	16	601	7	0
192.168.3.107	2	17	601	4	0
192.168.3.108	2	18	601	7	0
192.168.3.109	2	19	601	2	0
192.168.3.100	2	20	601	3	0
192.168.3.101	3	21	601	4	0
192.168.3.102	3	22	601	5	0
192.168.3.103	3	23	601	3	0
192.168.3.104	3	24	601	7	0
192.168.3.105	3	25	601	5	0

Summary

This chapter describes the testbed's integration of various technologies and the custom applications. The Cloud Chamber generated the data presented throughout this work. This testbed is based on just under 12,000 lines of code across all the custom applications. Millions of records were collected in each of the datasets presented in the next three chapters. The dataset's total record count in all four data collection tables from Chapter 6 was 5.9 million records. In the Chapter 6 experiment, *serviceman* returned

1.054 terabytes of data to the *loadrunners*. The system should scale up much larger given more hardware resources. This chapter concludes with the code used to consume the resources for each call to a service.

```
void consume(wservice * s, cputicks t0[], cputicks t1[], int count)

{
    char            *data;
    long            i, j;
    char            outbuffer[1025];
    int             size[RESOURCE_COUNT];
    time_t          now;
    char            logmsg[128];
    struct          timespec startburn;
    struct          timespec moment;
    long            howlongtoburn, diff;

    // update time marker
    time(&s->timelastseen);

    // get the amount of resources to consum from the distribution
    for(i=0;i<RESOURCE_COUNT;i++)
        size[i] = (int)getdistrovalue(s->distro[i], s->distro_arg[i][0], s->distro_arg[i][1], s->distro_arg[i][2]);

    if(LOG_LEVEL >=80 )
    {
        logg(80, "start - consume()");
        sprintf(logmsg, "service: %d  --> %d %d %d<p>", s->id,size[0],size[1],size[2]);
        logg(80,logmsg);
    }

    // start the stop watch
    gettimeofday(&startburn, NULL);
    clock_gettime(CLOCK_THREAD_CPUTIME_ID , &startburn);

    //---------------------------------------------------------------------
    // BANDWIDTH
    // --------------------------------------------------------------------
        // build a 1KB buffer for output
        for (i=0;i<1025;i++)    {               // loop for output
                outbuffer[i] = (char)((rand() % 94)+32);    // output string of 1024 chars
        }
        outbuffer[1025]=(char)0;                // terminate the string

        for (i=0;i<size[2];i++)    {            // loop for output
                printf("%s",outbuffer);                 // output string of 1024 chars
        }

        // throw into accumulator
        s->used[2] += size[2];

        // snooze just for a moment to let some data out the door
        struct timespec req, rem;
        req.tv_sec = 0;
        req.tv_nsec = 1;
        nanosleep(&req, &rem);

    //---------------------------------------------------------------------
    // MEMORY (short term)
    // --------------------------------------------------------------------
        data = (char*)malloc(size[1]);          // consume the memory

        for (i=0;i<size[1];i+=50)    {          // write to memory (every 50th address)
                data[i] = (char)((i % 255)+1);          // put something there, so OS will really give it to us
        }

        // throw into accumulator
        s->used[1] += size[1];

    //---------------------------------------------------------------------
    // MEMORY (persistent)
    // --------------------------------------------------------------------
        if (!s->data[1])
        {
            // malloc some data
            s->data[1] = (char*)malloc(size[1]);
            for (j=0;j<size[1];j+=50)
            {
                (s->data[1])[j] = (char)(rand() % 256);
            }
        }

    //---------------------------------------------------------------------
    // CPU
    // --------------------------------------------------------------------
        // capture the cpu consumption so far.
        clock_gettime(CLOCK_THREAD_CPUTIME_ID , &moment);
        diff = (moment.tv_sec-startburn.tv_sec)*1000000 + (moment.tv_nsec-startburn.tv_nsec)/1000;
        s->used[0] += (diff * noderesourcesvalues[0]);

        // reset sop watch and burn some cycles.
        clock_gettime(CLOCK_THREAD_CPUTIME_ID , &startburn);
        clock_gettime(CLOCK_THREAD_CPUTIME_ID , &moment);

        // this is adjusted to make slower cpus burn longer!! readjusted later before accumulating
        howlongtoburn = (long)(size[0] * (1.0 / noderesourcesvalues[0]));

        // calc the difference in microseconds
        diff = (moment.tv_sec-startburn.tv_sec)*1000000 + (moment.tv_nsec-startburn.tv_nsec)/1000;
        while(diff < howlongtoburn)
        {
            for (i=0;i<(moment.tv_sec % 1000);i++); // burn some cycles
            clock_gettime(CLOCK_THREAD_CPUTIME_ID , &moment);
            diff = (moment.tv_sec-startburn.tv_sec)*1000000 + (moment.tv_nsec-startburn.tv_nsec)/1000;
        }

        // readjust to reflect work units in terms of r = 1.0, see adjustment above on howlongtoburn
        s->used[0] += (diff * noderesourcesvalues[0]);

    free(data);

    s->timespent += (diff /1000);

    logg(80,"end - consume()");

}
```

IV. Provisioning Norm: A Placement Quality Measure to Balance Service Performance and Inclusion

Introduction

In a traditional operating system (OS), the kernel is responsible for identifying resources, interfacing with them, and managing their utilization. The OS prioritizes process utilization of the CPU. The OS manages virtual memory and pages to physical memory. The OS manages the send queue of the network interfaces. The OS manages these items with fairness in order for all processes to execute. In handling these resources, the OS abstracts the implementation which separates the details of the underlying hardware from the executing process.

A similar abstraction and management layer is needed in the platform as a service (PaaS) and software as a service (SaaS) computing models. In other words, to provide computing resources as a utility to paying consumers, the infrastructure must hide from the consumer the underlying physical and logical mechanisms. Consumers pay for an agreed upon level of service referred to as Quality of Service (QoS) in the form of Service Level Agreements (SLA) that are typically expressed in terms of response time or throughput; not typically in terms of number of servers, routers, cycles, etc. The abstraction allows the consumer to focus on developing their web service's business logic and user base free of system administration and operating system concerns in an agnostic fashion.

In this new, non-traditional distributed operating system, the "kernel" needs to identify physical resources as they become available, interface with them, and manage their utilization. These physical resources, typically processing, memory, bandwidth, and persistent storage, can be subdivided into virtual nodes as required, so the resources can be virtual or physical This “kernel” is aware of the services (processes) requiring resources. The “kernel” assigns, based on priority, policy, SLAs, and available resources, the service demands to the node supply. Once assigned, services and nodes are monitored for effective and efficient performance. As needed, this “kernel” will reassign services and nodes. As an example, in a recent Microsoft sponsored introductory technical report, Chappell [56] states that the recently introduced Windows Azure Fabric Controller creates the virtual machines required by an application, monitors the instances, starts new ones as needed, and patches operating system and other software. This reference shows that large corporate PaaS and SaaS vendors realize the need for more sophisticated service management.

For success in a large scale environment in which the web services are assigned to the web servers, configurations of assignments must be found quickly. In looking for proper configurations, performance and inclusion must be taken into account. A balance must be struck between performance and inclusion as resources come under stress. If performance is preferred, services will be excluded from configurations when insufficient resources are available. If inclusion is preferred, all services will be included, but performance will suffer. This chapter presents a tunable parameter allowing system administrators to explicitly implement their desired policy. This chapter addresses this in static scenarios, which is particularly useful for relatively stable services and affiliated loads.

This chapter introduces the Provisioning Norm. This chapter has revised and extended the original work [57]. This chapter presents a parameterized, mathematical measure assigning web services to servers based directly on resource requirements with a parameter addressing performance and inclusion requirements. The Provisioning Norm is presented here with formal, simulated, and empirical validation. Given a set of static services and static nodes, the Provisioning Norm quantifies web service placement quality, qualified by performance and inclusion requirements. This quantification of performance and inclusion is critical to managing web services autonomically in a large scale environment.

This chapter is presented in eight sections. The next section presents the mathematical model, introduces a way to quantify service resource requirements and server resources, and proves two theorems about the introduced Provisioning Norm. The third section quantifies performance and inclusion policy into a user-tunable parameter and proves three theorems regarding its implementation. Section four describes the testbed and simulation environments and presents experimental results. Section five discusses alternative quality measures. Section six, seven and eight are, respectively, related works, future work, and the chapter’s conclusion.

Quantification

The first part of this chapter establishes the means of quantifying the resources provided by servers (nodes), the resources required by services, and most importantly, the quality of an arrangement of the services on the servers (a configuration). This section first describes the matrix model used to represent services, servers, and their relationships. Using this model, matrix norms are shown to be inadequate means of quantifying the quality of services on servers. The Provisioning Norm is defined and shown to be an asymmetric norm [58][59]. The Provisioning Norm is shown to create a quantitative separation between good and bad service placements.

Matrix Model

Services require specific computational resources to function. These resources include, but are not limited to, CPU cycles, short term memory, long term storage, and bandwidth. As discussed in [5], these requirements can be seen as a type of workload profile for the service. The profile is a vector of numeric values indicating the service's resource requirements. Each element of the vector represents the required amount of a type of resource. Each entry is normalized to the interval [0,1]. For example, if services have five categories of resource requirements, then a service can be described using a five-valued vector,

<0.2 0.3 0.4 0.1 0.3>.

This chapter defines a service as an executable program on an individual computational node, i.e. one node can service many services. A more complicated service requiring multiple computational nodes is decomposed into separate (albeit dependent) services.

The service's resource requirements are fulfilled by the physical or virtual computational nodes upon which the services are executed. Similar to a service, a node is modeled as a vector of resources. The node's resource vector indicates the resources provided by the node to the system. The number of elements in the node vector is equal to the number of elements in the service vector; each corresponding element represents the same resource type.

The system is a set of nodes and services. The set of nodes is a matrix of the nodes' individual vectors with one row for each node. Similarly the services are modeled with a matrix of the service vectors. The mapping of services to nodes (or nodes to services) is captured in an adjacency matrix linking the supply of resources to the demand for resources. We define the service placement problem with the following question. What is the mapping that effectively executes the services, and how do we efficiently determine that mapping?

Three matrices formally model the problem. The node matrix, *N*, describes the resources *provided* by each node in the system. For example, a three column, twenty row *N* matrix models a system with three resources in twenty nodes. The service matrix, *S*, describes the resources *required* by each service in the system. For example, a three column, fifty row *S* matrix models a system with three resources in fifty services. The configuration (adjacency) matrix, *C*, describes how the services are assigned to the nodes. For example, a fifty column, twenty row *C* matrix models a system with twenty nodes and fifty services. The values in the node matrix *N* and the service matrix *S* are numerical values in the interval [0,1] representing the resources, respectively offered by the nodes and required by the services. Each column of these matrices represents a particular type of resource, such as processing power, memory, storage, and bandwidth. The configuration matrix *C* is an adjacency matrix. The c_{ij} entry indicates whether the i^{th} node provides resources to the j^{th} service. Services which are assigned to a node and collectively require more resources than the node provides are under-provisioned. Services which are assigned to a node and collectively require fewer resources than the node provides are over-provisioned.

Matrix Calculations

The total amount of resources consumed on a particular node is the product of *C* and *S*. The derived matrix contains a row for each node and a column for each aggregated resource. The c_{ij} entry in the derived matrix (5) indicates how much of the i^{th} resource of the j^{th} node is allocated in the current configuration;

$$CS\,. \tag{5}$$

The quality of this configuration is expressed as a single, non-negative, real-valued number. If $CS=N$, then the configuration *C* meets the demands of the services exactly: no service is under-provisioned and no service is over-provisioned. The amount of over- and under-provisioning is determined by the difference between the *CS* and *N* matrices (6), that is

$$N-CS\,. \tag{6}$$

Provisioning Norm

This section discusses the need for a measure of the quality of a configuration *C* for *N* and *S*. This section further proves the insufficiency of matrix norms as a quality measure. Finally, the Provisioning Norm is defined and proven to be a sufficient measure of configuration quality.

Definition 1 (Under-provisioned service) Given node matrix *N*, service matrix *S*, and configuration matrix *C*, an entry $x_{ij}<0$ in the matrix *N*-*CS* represents over-allocation of the i^{th} resource of the j^{th} node, i.e. services requiring the i^{th} resource of the j^{th} node are *under-provisioned* with respect to that resource.

Definition 2 (Over-provisioned service) Given node matrix *N*, service matrix *S*, and configuration matrix *C*, an entry $x_{ij}>0$ in the matrix *N*-*CS* represents under-utilization of the i^{th} resource of the j^{th} node, i.e. services requiring the i^{th} resource of the j^{th} node are *over-provisioned* with respect to that resource.

Definition 3 (Perfect Provision) Given node matrix *N*, service matrix *S*, and configuration matrix *C*, *N-CS* = **0** implies *C* is a perfectly provisioned configuration.

Definition 4 The configuration quality is the distance of *N-CS* from **0**.

Definition 5 Let *X* be a matrix space over the real field $\mathbb{R}$. A norm *p* is a function $p : X \to \mathbb{R}^+$ such that the following hold ture.

1. $\forall x \in X, p\left(x\right) \geq 0$

2. $x \in X, x = 0 \Leftrightarrow p\left(x\right) = 0$

3. $x \in X, \lambda \in \mathbb{R}^+, p\left(\lambda x\right) = \left|\lambda\right| p\left(x\right)$

4. $x, y \in X, p\left(x + y\right) \leq p\left(x\right) + p\left(y\right)$

The Frobenius norm is a matrix norm, as defined in Definition 5. The Frobenius norm, also known as the Euclidean norm, is defined as

$$\left\|A\right\|_F = \sqrt{\sum_i^m \sum_j^n \left|a_{ij}\right|^2} \quad . \tag{7}$$

The Frobenius norm reflects the Euclidean distance of a given matrix from the **0** matrix. By Definition 4 the Frobenius Norm is a quality measure of a configuration. Under- or over-provisioning increases as (8) increases.

$$\left\|N - CS\right\|_F \tag{8}$$

The Frobenius Norm as a quality measure fails to, and is incapable of, discriminating between under-provisioning and over-provisioning. The theorem and proof follow.

Theorem 1 Matrix norms do not discriminate between under-provisioning and over-provisioning.

Proof The symmetry property (Property 3 of Definition 5) of norms demands this. Over-provisioning is represented by positive values in the matrix. Under-provisioning is represented by negative values. Assume *M* is a matrix of all positive values, i.e. it is completely over-provisioned. For a matrix norm *p* and its symmetry property, $p(M) = p(-M)$. Therefore, matrix norms do not discriminate between under-provisioning and over-provisioning.

Services must be assigned to nodes with sufficient resources to meet their functional requirements. If the node does not have enough resources to properly execute the services assigned to it, i.e. the services on that node are under-provisioned, then the quality of the configuration cannot be described as acceptable quality. If all the nodes over-provision their services, i.e. all services have the resources they require and additional available resources remain, then the configuration has better quality than those with under-provisioning. This calls for a distance measure that biases under-provisioning.

Definition 6 The Provisioning Norm $\|M\|_{\alpha,F}$ is defined as $\|M\|_{\alpha,F} = (1-\alpha)\|M_{+}\|_{F} + \alpha\|M_{-}\|_{F}$ with $\alpha \in (0,1)$ such that

$$M = M_{+} + M_{-}$$

.

$$M_{+} \text{ with entries } m_{+ij} = \begin{cases} m_{ij}, m_{ij} \geq 0 \\ 0, m_{ij} < 0 \end{cases}$$

$$M_{-} \text{ with entries } m_{-ij} = \begin{cases} m_{ij}, m_{ij} \leq 0 \\ 0, m_{ij} > 0 \end{cases}$$

The Provisioning Norm is the sum of two biased Frobenius norms. The biased Frobenius norm of the positive entries of *M*, $(1-\alpha)\|M_{+}\|_F$, is added to the biased Frobenius norm of the negative entries of *M*, $\alpha\|M_{-}\|_F$.

Definition 7 (Asymmetric Norm) Let *X* be a matrix space. An *asymmetric* norm *p* such that $p: X \rightarrow \mathbf{R}$ with the following. Note the difference in Property 3 of this definition and Definition 5 Property 3.

1. $\forall x \in X, p(x) \geq 0$
2. $x \in X, x = 0 \Leftrightarrow p(x) = 0$
3. $x \in X, \lambda \geq 0, p(\lambda x) = \lambda p(x)$
4. $x, y \in X, p(x+y) \leq p(x) + p(y)$

Theorem 2 The Provisioning Norm is an asymmetric norm.

Proof Property 1 of an asymmetric norm is satisfied because the Provisioning Norm is the addition of two norms. By Definition 7, property 1, each of those norms is positive if there are any non-zero entries. The parameter α is by definition greater than 0. The sum of two positive values is a non-zero value.

Property 2 of an asymmetric norm is satisfied. If x=0, by Definition 7, Property 2, each of those norms is zero; the sum of zeros is zero. Therefore $x = 0 \Rightarrow \|x\|_{\alpha,F} = 0$. If $\|x\|_{\alpha,F} = 0$, by Definition 7, Property 1, both norms of the solution are zero or positive and the parameter α is by definition greater than 0. The sum can only be zero if both norms are zero, therefore x=0.

Property 3 holds based on the distributive property of multiplication over addition, Definition 7, Property 3, and by Definition 5, Property 2: $\lambda \geq 0$.

$$\left((1-\alpha)\|\lambda X_+\|_F + \alpha\|\lambda X_-\|_F\right) = \lambda\left((1-\alpha)\|X_+\|_F + \alpha\|X_-\|_F\right)$$
$$\|\lambda X\|_{\alpha,F} = \lambda\|X\|_{\alpha,F}$$

Property 4 can be derived from some observations and utilizing the triangular inequality property of a norm, Definition 7, Property 4 and the fact for A,B,C,D > 0 with $A < B$, $C < D$, the inequality $A + C < B + D$ holds. Let X, Y be two matrices of equal size and $Z = X + Y$. Let $X = X_+ + X_-$, $Y = Y_+ + Y_-$, and $Z = Z_+ + Z_-$. Where each is broken into positive entries in the + matrix and negative entries in the – matrix, see Definition 6.

Case I. From observations about entries in Z+ based on entries X and Y when both are positive, both negative, and when mixed, the following inequality holds. It is then manipulated into the appropriate norm definition. The norm definition is applied. And finally, Definition 7, Property 4, allows for a substitution.

$$0 \leq |z_{+ij}| \leq |x_{+ij} + y_{+ij}|$$
$$(1-\alpha)\left(\sum|z_{+ij}|^2\right)^{1/2} \leq (1-\alpha)\left(\sum|x_{+ij} + y_{+ij}|^2\right)^{1/2}$$
$$(1-\alpha)\|[X+Y]_+\|_F \leq (1-\alpha)\|X_+ + Y_+\|_F$$

Case II. Similar observations about Z- lead to the following inequality.

$$0 \le \left|z_{-ij}\right| \le \left|x_{-ij} + y_{-ij}\right|$$

$$\alpha\left(\sum\left|z_{-ij}\right|^2\right)^{1/2} \le \alpha\left(\sum\left|x_{-ij} + y_{-ij}\right|^2\right)^{1/2}$$

$$\alpha\left\|\left[X + Y\right]_-\right\|_F \le \alpha\left\|X_- + Y_-\right\|_F$$

Finally, the inequalities derived in Case I and II are added together.

$$(1-\alpha)\left\|\left[X + Y\right]_+\right\|_F + \alpha\left\|\left[X + Y\right]_-\right\|_F \le (1-\alpha)\left\|X_+ + Y_+\right\|_F + \alpha\left\|X_+ + Y_+\right\|_F$$

The left side becomes $\left\|X + Y\right\|_{\alpha,F}$ using the definition of the Provisioning Norm. The right side of the inequality becomes $\left\|X\right\|_{\alpha,F} + \left\|Y\right\|_{\alpha,F}$ the following after using the triangular inequality property of norms (Definition 7, property 4), the distributive property of multiplication over addition, and finally the definition of the provisioning norm. Thus,

$$\left\|X + Y\right\|_{\alpha,F} \le \left\|X\right\|_{\alpha,F} + \left\|Y\right\|_{\alpha,F}$$

Having shown that the triangular inequality of the Provisioning Norm holds, the Provisioning Norm is an asymmetric norm.

Service Performance and Inclusion

Best case performance and guaranteed inclusion exclude each other in any system experiencing a non-trivial load. As the systems encounter heavy loads policy decisions direct the systems to handle the load. The policies can be as simple or complex as needed.

One extreme requires guaranteed performance. In this case, in order to maintain the high level of performance clearly some services will be entirely excluded. Alternatively, the other extreme requires guaranteed inclusion. In this case, in order to maintain the high level of inclusion, the performance of all services suffers. In between these extremes, volumes of works have been developed to manage priorities, exceptions, costs, and profits.

This section discusses theorems and formulas to quantify policy directives into a parameter. The Provisioning Norm uses this parameter to properly balance performance and inclusion. Parameter values near zero favor inclusion over performance. Parameter values near one favor performance over inclusion. Values in between allow for a proper balancing of these needs. Simulations and numeric methods are presented to determine a proper setting for α given a desired number of services to include.

In this chapter, performance is assumed to be tied to properly provisioned configuration. Empirical investigation of this is in a later section. Inclusion is assumed to be the number of services included. There is no concept here of prioritization of service inclusion. The Provisioning Norm is agnostic to the concepts of priority and profit/cost. The heuristic used in this dissertation includes and excludes a specific service based strictly on the quality of the configuration. The assumption for this dissertation is service priority and profit are engineered in another mechanism. For example, the heuristic searching for a new configuration probabilistically considers removing a service from a node or adding available services to nodes. Queues, weights, and other potential mechanisms can be employed in conjunction with the Provisioning Norm. Priority and profit are considered to be constraints that are applied to the system from the outside.

Future work can address potentially integrating priority and profit into the mathematical model. However, this dissertation considers specifically measuring the quality of the configuration and its relationship to performance and inclusion.

Qualifying Performance

The following two theorems assert that there exists a parameter value such that configurations (services assigned to nodes) *with no* under-provisioning have lower Provisioning Norms than configurations *with* under-provisioning. The Provisioning Norm creates a partial ordering over the configurations. Heuristic and numeric methods can be employed to find configurations with no under-provisioning.

Theorem 3. Let *N* be an *n*×*r* matrix of node resources with *n* nodes (rows) and *r* resources (columns), *S* be an *s*×*r* matrix of service resources with *s* services (rows) and *r* resources (columns), *C* be a *s*×*n* adjacency matrix mapping services to nodes. Let $\mathbb{C}$ be the set of all possible *s*×*n* adjacency matrices. Entries in *N* and *S* are in the interval [ε,1] where ε is the smallest resource representation (e.g. resolution). For notational purposes let the Provisioning Norm be noted as

$$q(C,N,S,\alpha)=\left\|N-CS\right\|_{\alpha,F}=(1-\alpha)\left\|\left[N-CS\right]_{+}\right\|_{F}+\alpha\left\|\left[N-CS\right]_{-}\right\|_{F}.$$

There exists an α such that

$$\frac{\sqrt{n\cdot r-1}}{\varepsilon+\sqrt{n\cdot r-1}}\geq\alpha>1$$

and

$$\max_{C\in\mathbb{C}^{+}_{N,S}} q(C,N,S,\alpha)<\min_{C\in\mathbb{C}^{-}_{N,S}} q(C,N,S,\alpha)$$

where

$$\mathbb{C}^{+}_{N,S} = \left\{ C \in \mathbb{C} : \forall ij, \left[N - CS \right]_{ij} \geq 0 \right\}$$
$$\mathbb{C}^{-}_{N,S} = \left\{ C \in \mathbb{C} : \exists ij, \left[N - CS \right]_{ij} < 0 \right\}$$

Proof $\mathbb{C}^{+}_{N,S}$ is the set of configuration for a fixed *N* and *S* such that there are no negative entries in the residual matrix $N - CS$. $\mathbb{C}^{-}_{N,S}$ is the set of configurations for the same fixed *N* and *S* such that there is at least one negative entry in the residual matrix $N - CS$. Let *max()* of a <u>norm </u>be the maximum value the norm takes on for a set of finite inputs. Let *max()* of a <u>*matrix*</u> be the largest entry in the matrix. Similarly define *min()* for both a norm and a matrix. Let *M* = *N-CS* for notational brevity. The Provisioning Norm of *M* is defined above as the sum of two norms. A lower bound of the α parameter, $\frac{\sqrt{n \cdot r - 1}}{\varepsilon + \sqrt{n \cdot r - 1}}$, is derived such that the norm of the *positive* matrix will always be less than the norm of the *negative* matrix times α. Consider the worst quality configuration with no negative entries $\max_{C \in \mathbb{C}^{+}_{N,S}} q\left(C, N, S, \alpha\right)$. Consider the best quality configuration with at least one negative entry $\min_{C \in \mathbb{C}^{-}_{N,S}} q\left(C, N, S, \alpha\right)$. And thus to ensure the best configuration with negative entries is guaranteed is of worse quality than all configurations with only positive entries, the following must hold

$$\max_{C \in \mathbb{C}^{+}_{N,S}} q\left(C, N, S, \alpha\right) < \min_{C \in \mathbb{C}^{-}_{N,S}} q\left(C, N, S, \alpha\right).$$

The following is then true and shortened for notional purposes.

$$\begin{aligned}M &= N - CS\\ \|M\|_{\alpha,F} &= (1-\alpha)\|M_+\|_F + \alpha\|M_-\|_F\\ (1-\alpha)\|M_+\|_F &< \alpha\|M_-\|_F\end{aligned}$$

Consider the inequality of the largest possible value of the norm of the positive matrix and the smallest possible non-zero value of the norm of the negative matrix. The smallest value is a positive number, so divide both sides by that number to isolate α.

$$(1-\alpha)\max\left(\|M_+\|_F\right) < \alpha\min\left(\|M_-\|_F\right)$$

$$\frac{\max\left(\|M_+\|_F\right)}{\min\left(\|M_-\|_F\right)} < \frac{\alpha}{(1-\alpha)}$$

Assume ε is the smallest possible value used to represent a resource value. The minimum non-zero value of the Frobenius norm of the negative M_- matrix is

$$\min\left(\|M_-\|_F\right) = \varepsilon .$$

This implies a single entry of $-\varepsilon$ in M_- and all other entries are 0.

Independently, the maximum value of the Frobenius norm would come from a matrix with the largest possible entries. Consider the case where S=**0** (all zero entries) and thus CS=**0**. The positive entries are $M_+ = M = N - \mathbf{0} = N$. The maximal node matrix has all entries of 1. The maximum Frobenius Norm of the positive entries is

$$\max\left(\|M_+\|_F\right) = \sqrt{n \cdot r} .$$

If there is minimal non-zero under-provisioning, then M_- is all zeros with a single entry of -ε. This implies M_+ is all ones with a zero entry corresponding to the -ε entry in M_-. Consider the small example matrix

$$M = \begin{vmatrix}1 & -\varepsilon\\ 1 & 1\end{vmatrix} \Rightarrow M_+ = \begin{vmatrix}1 & 0\\ 1 & 1\end{vmatrix}, M_- = \begin{vmatrix}0 & -\varepsilon\\ 0 & 0\end{vmatrix}.$$

The adjusted maximum Frobenius Norm of the positive matrix accounts for the implied 0 entry and for M_+ with $M_{+,ij}$ in the interval [0,1] is

$$\max\left(\left\|M_+\right\|_F\right) = \sqrt{\left(n \cdot r - 1\right)} .$$

The derived lower bound ensures the dominance of the negative matrix over the positive matrix in determining the value of the Provisioning Norm. Thus,

$$\frac{\max\left(\left\|M_+\right\|_F\right)}{\min\left(\left\|M_-\right\|_F\right)} < \frac{\alpha}{\left(1-\alpha\right)}$$

with substitutions reduces to

$$\alpha > \frac{\sqrt{\left(n \cdot r - 1\right)}}{\varepsilon + \sqrt{\left(n \cdot r - 1\right)}} .$$

In practical terms, Theorem 3 states that with the properly selected parameter, all configurations with any under-provisioning are guaranteed to have a worse quality measure (higher) than all configurations with no under-provisioning. Theorem 4 provides a formula to determine what specific value the Provisioning Norm must be below to know there are no under-provisioned services.

Theorem 4. Theorem 3 (α is sufficiently large) implies Provisioning Norm values less than $\alpha \cdot \varepsilon$ have no under-provisioning.

Proof Assume Theorem 3 and its assumptions. For configurations with no under-provisioning: *M*=M_+ and M_-=**0**. The smallest possible Provisioning Norm with under-provisioning is *M*=M_-, where M_- is all zeros with a single entry of ε and M_+=**0**. With α selected per Theorem 3, the following holds.

$$\left(1-\alpha\right)\max\left(\left\|M_{+}\right\|_{F}\right) < \alpha \min\left(\left\|M_{-}\right\|_{F}\right)$$

And substituting ε for $\min\left(\left\|M_{-}\right\|_{F}\right)$,

$$\left(1-\alpha\right)\max\left(\left\|M_{+}\right\|_{F}\right) < \alpha \cdot \varepsilon \, .$$

To summarize the two previous theorems, if the policy parameter for the Provisioning Norm α is selected sufficiently large, $\frac{\sqrt{n \cdot r - 1}}{\varepsilon + \sqrt{n \cdot r - 1}} \geq \alpha > 1$, then any configuration with a Provisioning Norm less than $\alpha{\cdot}\varepsilon$ is guaranteed to have no under-provisioning, e.g. no nodes are over capacity.

Qualifying Inclusion

The theorem presented below asserts that if the Provisioning Norm parameter is set low enough and there is not gross over demand for node resources, then all services will be available. If there is a gross over demand for resources, then as many services as possible are made available.

The Provisioning Norm with a sufficiently small α cannot guarantee in all cases to include all services. This is due to the underlying mathematical behavior of the Provisioning Norm. For example, assume the degenerative case where the services demand 10 times the amount of resources the nodes provide. The configurations with the lowest valued Provisioning Norm (with a small α) will be those in which all node resources are assigned (no over-provisioning exists) and under-provisioning is minimized. Assigning additional services to such configurations increases under-provisioning and the value of the Provisioning Norm.

Theorem 5. Let *N* be an *n*×*r* matrix of node resources with *n* nodes (rows) and *r* resources (columns), *S* be an *s*×*r* matrix of service resources with *s* services (rows) and *r* resources (columns), *C* be a *s*×*n* adjacency matrix mapping services to nodes. Let $\mathbb{C}$ be the set of all possible *s*×*n* adjacency matrices. Entries in *N* and *S* are in the interval [*ε*,1] where *ε* is the smallest resource representation (e.g. resolution). For notational purposes let the Provisioning Norm be noted as

$$q(C,N,S,\alpha)=\|N-CS\|_{\alpha,F}=(1-\alpha)\left\|[N-CS]_{+}\right\|_{F}+\alpha\left\|[N-CS]_{-}\right\|_{F}.$$

There exists an α such that

$$0<\alpha\leq\frac{\varepsilon}{\varepsilon+\sqrt{(n\cdot r-1)}}$$

and

$$\max_{C\in\mathbb{C}_{N,S}^{-}} q(C,N,S,\alpha)<\min_{C\in\mathbb{C}_{N,S}^{+}} q(C,N,S,\alpha)$$

where

$$\mathbb{C}_{N,S}^{+}=\left\{C\in\mathbb{C}:\forall ij,[N-CS]_{ij}\geq 0\right\}$$
$$\mathbb{C}_{N,S}^{-}=\left\{C\in\mathbb{C}:\exists ij,[N-CS]_{ij}<0\right\}$$

Proof. The sufficiently small value of α is derived from a modified version of Theorem 3 proof.. The value of α such that

$$0<\alpha\leq\frac{\varepsilon}{\varepsilon+\sqrt{(n\cdot r-1)}}$$

numerically separates configurations with over-provisioning from those with no over-provisioning. Configurations with even a single entry of over-provisioning in M_+ have a significantly higher provisioning norm because 1-α >> α. In other words, leaving resources available on a node is mathematically less preferable. Therefore as the heuristic searches for configurations of good quality, filling nodes mathematically improves the quality, i.e. reduces the Provisioning Norm. This process continues adding services to nodes by filling unused resources on nodes even if other resources are over-provisioned because the mathematical benefit of using unused resources is much greater than over-provisioning. If all services are not assigned and all of the resources have been assigned, e.g. $M_+=\mathbf{0}$, additional services will not be assigned because the provisioning norm increases.

Note the subtle inversion in $\max_{C\in\mathbb{C}^-_{N,S}} q\left(C,N,S,\alpha\right)<\min_{C\in\mathbb{C}^+_{N,S}} q\left(C,N,S,\alpha\right)$ from Theorem 3. Using the sufficiently low α makes the best (min) quality configuration resulting in a matrix with a *positive* value worse than the worst (max) with no *positive* entry. Note the following similarly derived in Theorem 3:

$$\min\left(\left\|M_+\right\|_F\right)=\varepsilon$$

$$\max\left(\left\|M_-\right\|_F\right)=\sqrt{n\cdot r-1}$$

$$\left(1-\alpha\right)\max\left(\left\|M_-\right\|_F\right)<\alpha\min\left(\left\|M_+\right\|_F\right)$$

$$\frac{\max\left(\left\|M_-\right\|_F\right)}{\min\left(\left\|M_-+\right\|_F\right)}<\frac{\left(1-\alpha\right)}{\alpha}$$

Substituting and reducing yields,

$$\alpha \le \frac{\varepsilon}{\varepsilon + \sqrt{(n \cdot r - 1)}}.$$

$$\sqrt{\qquad\qquad}$$

Range of α

MATLAB simulations were executed to demonstrate the assignment behavior of the Provisioning Norm as the parameter α varies. The assignment behavior causes over-provisioned resources and/or under-provisioned resources as well as inclusion of some or all the services. The investigation looks to model this behavior in four separate scenarios. The Low Service Demand-High Node Supply represents the trivial case where plenty of node resources are available to supply the service demand. The Low Medium Service Demand-Medium Node Supply is a case where the supply is sufficient but with minimal over-provisioning. The Medium Service Demand-Low Medium Node Supply case provides insight when supply is insufficient but with minimal under-provisioning. The High Service Demand-Medium Node Supply case overwhelms the supply with significant excess demand. In each case, the number of under-provisions, over-provisions, and included services is plotted over values of α.

These simulations created arbitrary node and service matrices, 25 nodes and 100 services. Using these node and service matrices, α was varied over the open interval (0,1) in increments of 0.025. For each iteration of α, the heuristic was executed 20 times; and the mean number of services assigned, the mean number of under-provision occurrences, and the mean number of over-provision occurrences were collected.

From these results, four representative examples are consolidated into Figures 5, 6, 7 and 8. The first example has the services' resource demand substantially less than the nodes' resource supply. The second has the services' resource demand less than the nodes' resource supply. The third has the services' resource demand essentially equal to the nodes' resource supply. The fourth has the services' resource demand substantially more than the nodes' resource supply. Each graph shows the services included, under-provisions, and over-provisions with dashed line showing two standard deviation intervals. Note that there were 75 possible node resources available for over- or under-provisioning (3 resources on 25 nodes).

Figure 6 shows that all services were assigned across all values of α. At lower values, most of the time there was no under-provisioning. This is to be expected when the total demand (sum of all entries in S) is 8.89 and the total supply (sum of all entries in N) is 28.72.

Figure 7 (demand=12.61, supply=18.43) shows that instances of under-provisioning starts at 20 with a low α, begins descending near α=0.4, and approaches zero at α=0.975. All services are included every time for α<0.75. For α>0.75, the number of services begins to decrease from 100 in order to minimize under-provisioning.

Figure 8 (demand=19.62, supply=17.35) shows α<0.25; all services are included with about 50 instances of under-provisioning. The number of included services declines steadily as α increases to where 70 included services show no under-provisioning.

Figure 9 (demand=42.52, supply=25.91) shows the case where under no conditions are 100 services included. Most services were included at the lowest α, and nearly all 75 node resources were under-provisioned. The under-provisioning was minimized at the highest α, but that required assigning only approximately 45 services.

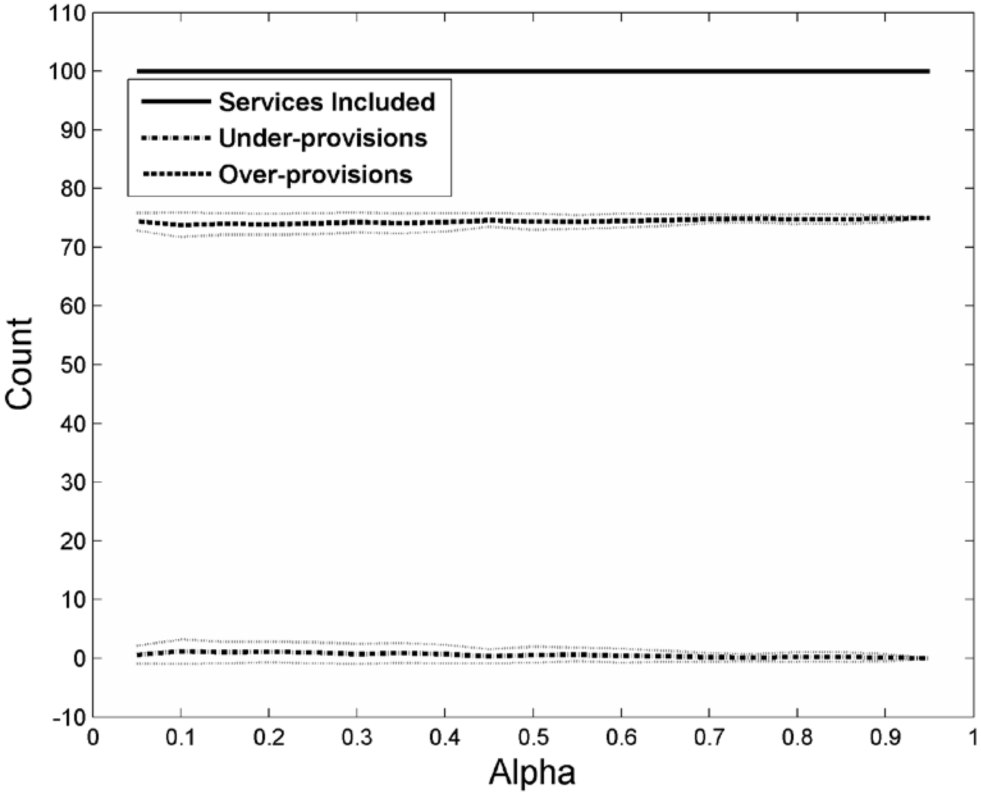


Figure 6. Services included, under-provisions, and over-provisions across α in a low resource demand. Total service resource demand = 8.98. Total node resource supply = 28.72.

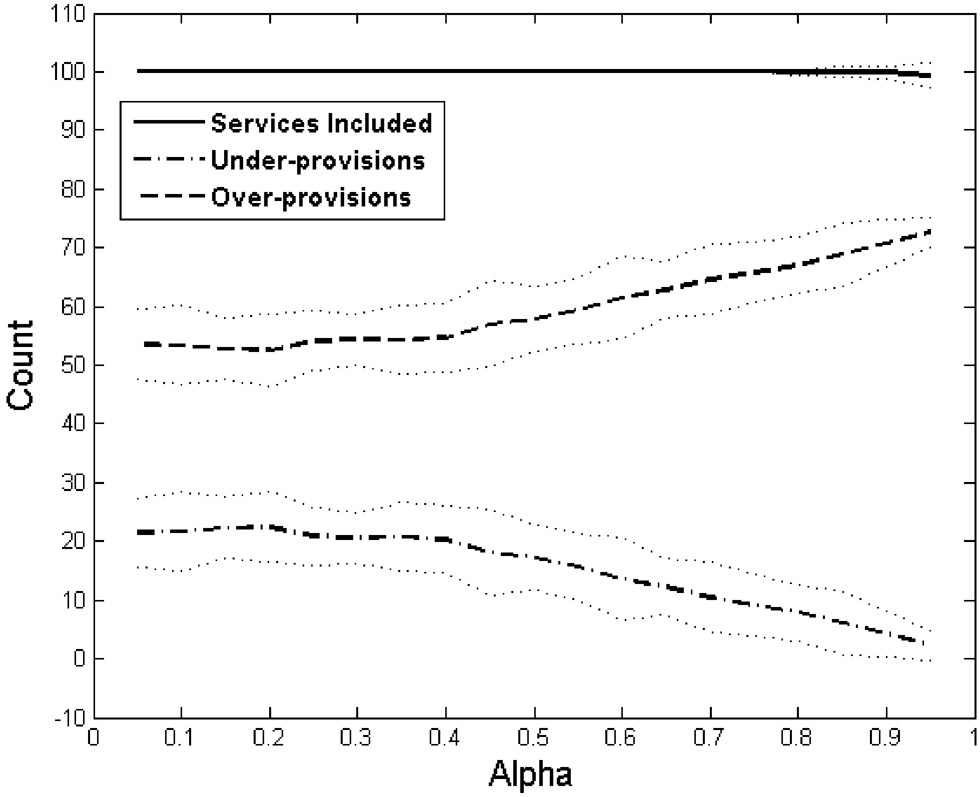


Figure 7. Services included, under-provisions, and over-provisions across α in a low-medium resource demand. Total service resource demand = 12.61. Total node resource supply = 18.43.

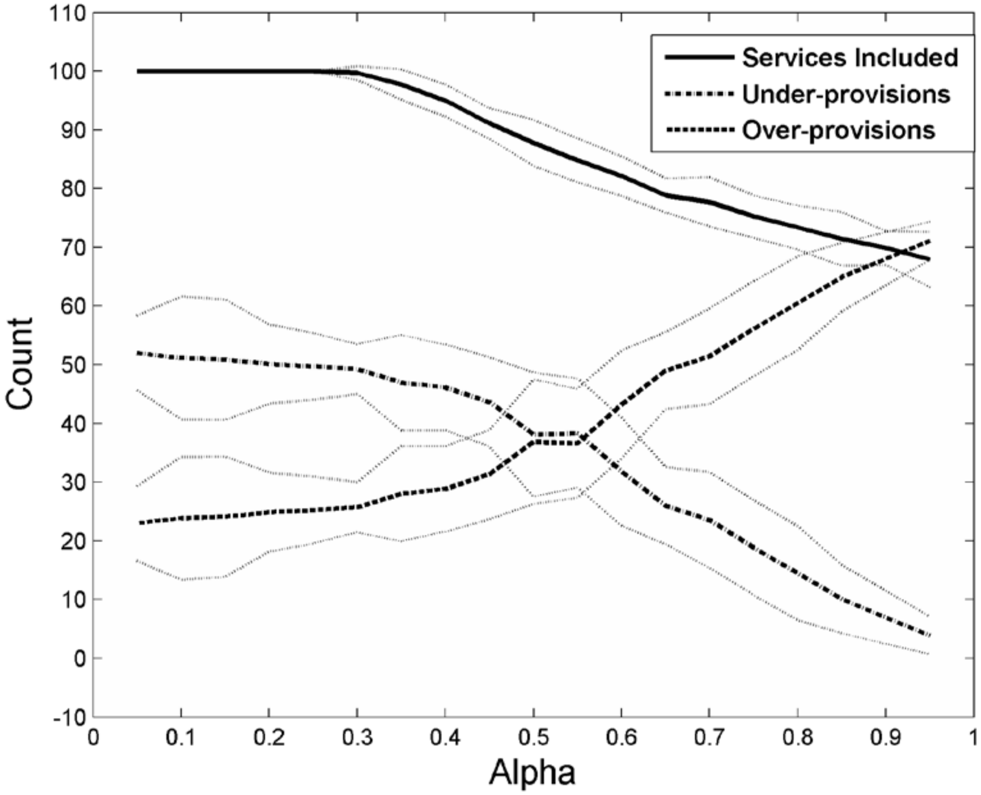


Figure 8. Services included, under-provisions, and over-provisions across α in a medium resource demand. Total service resource demand = 19.62. Total node resource supply = 17.35.

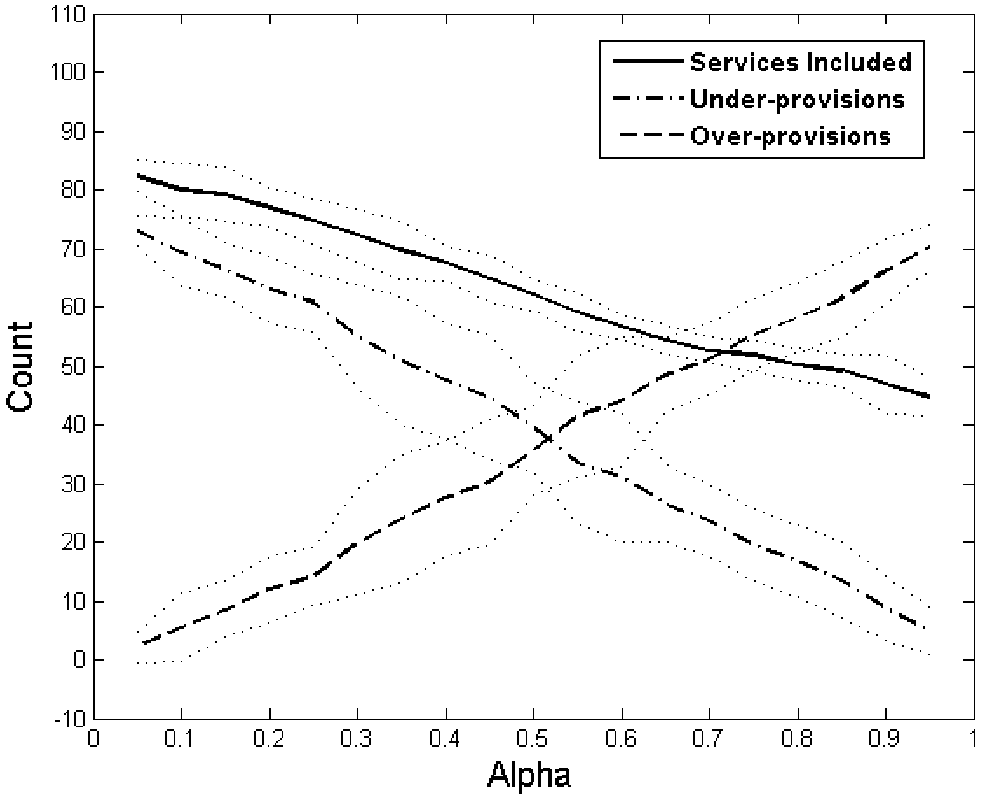


Figure 9. Services included, under-provisions, and over-provisions across α in a high resource demand. Total service resource demand = 42.52. Total node resource supply = 25.91.

Numeric Methods for α

As expected, the graphs from the previous section show a clear relationship between the values of α and the number of services included. The goal of this section is to show that a specified target number of services will be included in the configuration. This requires using the relationship between the included services and α. Two approaches using a linear approximation and a secant method are outlined below. The linear approximation provides the fastest possible approximation with less precision. The secant method is relatively slower but provides a more accurate solution. Due to the complications created by the large number of possible node and service matrices, formulating a precise, simple calculation of the expected number of included services given α, or vice versa, is a challenge not directly addressed in this dissertation.

Quick Method

The linear approximation is presented first. Its utility trades speed for accuracy. This approach requires executing the search heuristic only twice and calculating a linear function between the two. The algorithm, in summary, finds the number of services included for α near 0 and for α near 1. These two points define a line. The line provides a function to find the estimated α required from a desired number of included services.

1. Calculate α* based on Theorem 3.

2. Select an initial configuration: the current configuration (if one exists), an arbitrary configuration, or an empty configuration.

3. Execute the heuristic twice using α* and 1-α*, respectively.

4. For α*, calculate s_1, the number of services included as the sum of the adjacency matrix C, c_{ij} is an entry in C,

$$s_1 = \sum_i \sum_j c_{ij} \quad .$$

5. Similarly calculate s_0 using the configuration determined by 1-α*. These derived values determine two linear functions of α.

$$s = (s_1 - s_0)\alpha + s_0$$

6. This linear approximation can be used to derive from the parameter α the services expected to be included and the expected under-provisioning. To implement a service inclusion policy, calculate α from the target number of included services is

$$\alpha = \frac{s - s_0}{s_1 - s_0} \, .$$

Note on the end points: if the desired $s > s_0$ then α=1- α*, and if the desired $s < s_1$ then α=α*.

Note on under-provisioning: the above method works similarly for implementing an under-provisioning policy. The sum and linear function for under provisioning are:

$$u_1 = \sum_i \sum_j x_{ij} \left| x_{ij} = \begin{cases} 1 if \left(N - CS\right)_{ij} < 0 \\ 0 if \left(N - CS\right)_{ij} \geq 0 \end{cases} \right.$$

$$u = (u_1 - u_0)\alpha + u_0$$

$$\alpha = \frac{u - u_0}{u_1 - u_0}$$

Note on expected error: this method clearly will have error in the result. In the simulations, it has been observed the included services graphs are generally linear; see Figure 6, 6, 7, and 8. Figure 8 is the worst in regard to linearity. This works well when the system is relatively stable. The error after several iterations can settle to something small. In a dynamic environment, the error would continue to be substantial each time. However, the error may be small enough to still be useful and was quick to acquire. In a dynamic environment, it may not be useful to spend time getting a more accurate number when that more accurate value may not be valid long enough to be useful. In this example if the target number of services is 90, this fast linear method returns α=0.3 when in reality it is closer to 0.4. The intent of this method is speed over accuracy. Accuracy is addressed in the following section.

Secant Method

The secant method [60] allows for a more accurate result, although it requires iterative executions of the search heuristic. The secant method is a numeric method used to obtain the zero(s) of a function when the empirical data is available but the actual function or formula is not known. The secant method is an iterative method in which the n+1 iteration produces the x_{n+1} such that $f(x_{n+1})$ is closer to zero than the previous $f(x_n)$. In our application, x is the α of interest. The secant method seeks to find α where $f(\alpha)=0$. The general secant method formula is

$$\alpha_{n+1} = \alpha_n - f\left(\alpha_n\right)\left[\frac{\alpha_n - \alpha_{n-1}}{f\left(\alpha_n\right) - f\left(\alpha_{n-1}\right)}\right].$$

Define $h(\alpha)$ as the number of services included in the configuration returned by the search heuristic on the Provisioning Norm with α, e.g. the sum of the adjacency matrix C, c_{ij} is an entry in C,

$$h(\alpha) = \sum_i \sum_j c_{ij} \ .$$

Define the function $f(\alpha)$ as $h(\alpha)$ minus the target number of services,

$$f(\alpha) = h(\alpha) - s \ .$$

1. Calculate α^* based on Theorem 3.

2. Select an initial configuration: the current configuration (if one exists), an arbitrary configuration, or an empty configuration.

3. Execute the heuristic twice using α^* and $1-\alpha^*$, respectively. To initialize the method, $\alpha_0= 1-\alpha^*$ and $\alpha_1= \alpha^*$.

4. Select the two halting conditions for the secant method, ε and δ. One condition is if the number of services found is close enough to the desired amount. The second is if α did not move a sufficient distance. The method terminates when either of these holds,

$$\begin{array}{r} f(\alpha_{n+1}) < \varepsilon \\ |\alpha_{n+1} - \alpha_n| < \delta \end{array} .$$

5. Iteratively calculate α_n+1 until one of two stopping conditions occurs.

Note on the expected error: A formal analysis of the error is not presented here. Finding zeros with the secant method for a known function is well documented [60]. The more interesting consideration in this application is that the heuristic returns configurations having a number of assigned services within a confidence interval. With sufficiently small selections of ε and δ, the primary source of any error will be from the confidence interval.

Empirical Methodology

This section described the testbed used to empirically demonstrate the utility of the above theorems. The testbed's physical and virtual components are described. These resources on these nodes are normalized. The profiling of the service resources and their normalization are presented. Using these profiles, a catalogue of configurations is derived and used to show how the varying quality of configurations (Provisioning Norm) changes performance with performance, e.g. does performance correlate with quality.

Testbed

The testbed described in Chapter 3, the Cloud Chamber [16], provides an environment for the placement of services on node resources. Clients execute services based on predefined loads. These loads produce performance data for both the services and the nodes. This data, using on-line analysis, determines a new service-to-node configuration. A centralized configuration manager implements a configuration by reassigning services to nodes.

To establish the node and service matrices, the nodes and services must be profiled. The profile is a numeric representation of resources. A node's profile is based on its resource capacities. A service's profile is based on its resource consumption. The profile results from the quantification of resource measurements into a range of normalized values. The profiling performed in the chapter assumes that the different resources are to be weighted equally. In order to accomplish this, the normalized value of each resource belongs to [0,1]. For the nodes, the CPU MHz ranges up to 1000 MHz, the memory ranges up to 512 MB, and bandwidth ranges up to 100 Mbps. A node with resources 1000 MHz, 512 MB, and 100 Mbps is profiled as <1.0, 1.0, 1.0>. A node with resources 500 MHz, 128 MB, and 100 Mbps is profiled as <0.5, 0.25, 1.0>.

Services are similarly profiled, either statically or dynamically. In the static method, each service is independently placed on a single node. The *loadrunner* issues a specified number of requests per second to exercise each service. The database collects the reported resource consumption of the node while the service is exercised. Finally the usage is translated into the normalized service profile. In the dynamic method, as the services are executed in the experiment, the amount of CPU, memory, and bandwidth consumption are measured. This allows for the profile to change over time. The second method is more interesting and preferable to account for on-line behavior.

For the next consideration, the node used for service profiling has the profile of <1.0, 1.0, 1.0>, meaning 1000MHz, 512MB, and 100Mbps. To simplify the experiment, the load is fixed at 20 requests per second. Each service is, one by one, exercised at 20 hits per second. The database collects the node's CPU usage, memory consumption, and bandwidth consumption. The service CPU resource value derives from the mean CPU utilization. The service memory resource value derives from (mean (Used memory – Used memory at t_0) / 512). The service bandwidth resource value derives from (mean (Bytes Sent * 8 bits) / 100,000,000.

The profiling described above yielded the following matrices in Table 6. Note S^* is 25 of the 100 services. Note also, with ε=0.0001 these numbers carry four significant digits.

Table 6. Sample Node and Service Resource Profile Matrices

$$N=\left|\begin{array}{ccc}
\text{cpu} & \text{mem} & \text{band} \\
1.00 & 0.45 & 0.10 \\
1.00 & 0.45 & 1.00 \\
0.50 & 0.90 & 0.10 \\
0.50 & 0.90 & 0.01 \\
0.50 & 0.90 & 0.10 \\
0.50 & 0.90 & 0.01 \\
0.50 & 0.22 & 1.00 \\
0.50 & 0.22 & 0.10 \\
0.50 & 0.22 & 0.10 \\
0.50 & 0.22 & 0.01 \\
0.25 & 0.22 & 0.01 \\
0.25 & 0.22 & 0.01 \\
0.25 & 0.22 & 0.01 \\
0.25 & 0.45 & 0.01 \\
0.25 & 0.45 & 1.00 \\
0.25 & 0.45 & 1.00 \\
0.25 & 0.45 & 0.10 \\
0.25 & 0.45 & 0.10 \\
0.25 & 0.22 & 0.10 \\
0.25 & 0.22 & 0.10 \\
0.25 & 0.90 & 0.10 \\
0.25 & 0.22 & 0.10 \\
0.25 & 0.22 & 0.10 \\
0.25 & 0.45 & 0.10 \\
1.00 & 0.90 & 1.00
\end{array}\right| \qquad
S^{*}=\left|\begin{array}{ccc}
\text{cpu} & \text{mem} & \text{band} \\
0.0724 & 0.0597 & 0.0017 \\
0.1089 & 0.0151 & 0.0083 \\
0.0965 & 0.0102 & 0.0017 \\
0.0762 & 0.1962 & 0.0017 \\
0.0844 & 0.0200 & 0.0165 \\
0.1157 & 0.0105 & 0.0165 \\
0.0554 & 0.0153 & 0.0017 \\
0.0713 & 0.0104 & 0.0017 \\
0.0589 & 0.0103 & 0.0083 \\
0.1758 & 0.0098 & 0.0821 \\
0.0648 & 0.0103 & 0.0017 \\
0.0432 & 0.0100 & 0.0017 \\
0.0963 & 0.0593 & 0.0165 \\
0.0572 & 0.0153 & 0.0017 \\
0.0813 & 0.0103 & 0.0165 \\
0.0511 & 0.0152 & 0.0083 \\
0.0858 & 0.0202 & 0.0017 \\
0.2183 & 0.0100 & 0.1231 \\
0.0471 & 0.0104 & 0.0083 \\
0.3079 & 0.0155 & 0.3281 \\
0.0683 & 0.0104 & 0.0017 \\
0.0719 & 0.0101 & 0.0083 \\
0.0719 & 0.0107 & 0.0083 \\
0.0782 & 0.0102 & 0.0165 \\
0.0927 & 0.0199 & 0.0017
\end{array}\right|$$

Configuration Quality and Performance

This section describes the process used to collect a catalog of configurations (using *N* and *S* above) and each corresponding quality measure. The quality of each configuration is the value of the Provisioning Norm of *N-CS*. This catalog has configurations with quality measures ranging from bad configurations (highly under-provisioned) to good configurations (not under-provisioned). Once the catalog was generated, configurations were selected to be executed in the testbed in order to compare the quality of the configuration to its actual performance.

The set of all possible configurations is huge. With 100 services assigned to one of 25 nodes, a space of 25^{100} configurations is too expensive for an exhaustive search to be practical. The first investigation sampled the quality measure of various configurations for the selected *N* and *S*. Ten million random configurations in which every service was assigned to a node were selected from the possible 25^{100} configurations. For each selected configuration the Provisioning Norm was calculated. Figure 10 presents the histogram of the results. The smallest found was approximately 0.1759. The largest was 1.8893. Per Theorem 4 with ε=0.0001 and α=0.99999, configurations with values less than 0.00099999 have no under-provisioning. In 10^6 samples, not one configuration was close to this value.

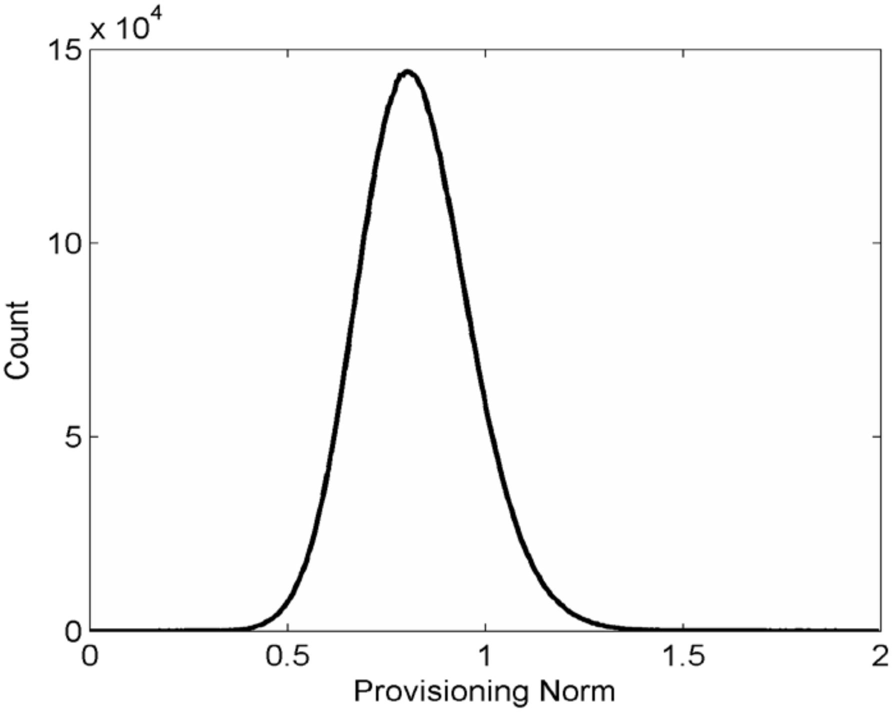


Figure 10. Histogram of 10 million sample configurations' provisioning norms.

In order to find configurations of better quality, a search heuristic is utilized. The method is described in [61]. The method is similar to the GRASP method [62] and the Multi-Start Hybrid method [63]. Generally, the method is a multi-start, local, random, greedy search method returning the best (lowest) configuration found after a specified number of rounds. Each round finds a better configuration by randomly selecting a node, removing the service (if any) in which the node is better off, and adding an available service (if any) in which the node is better off. Better off means the Provisioning Norm for that node alone is smaller (better). Using *N* and *S* from the previous section, an example of this descent is in Figure 11. Using the Provisioning Norm with $\alpha = 0.99999$, the initial random configuration's quality was 0.7839. The final configuration's quality settles at 0.00003064 after 150 rounds.

To create a catalog of configurations, a collection of 1000 arbitrary configurations were initially selected, i.e. services were randomly assigned to nodes. Using the Provisioning Norm with $\alpha = 0.99999$ and the heuristic, each configuration yielded a configuration of high quality. At each round of all 1000 heuristic runs, the configuration and its quality measure were logged to a database. An example of the heuristic decent toward the best configuration is depicted in Figure 11. Each drop represents a better configuration. Each of these new found configurations and its quality are logged to the database. A total of 59,767 configurations were cataloged.

A total of 3585 of the 59,767 configurations had quality measures less than $\alpha\varepsilon = 0.000099999$. Recall from Theorem 4, a quality measure less than $\alpha\varepsilon$ (with α sufficiently large) indicates configurations with no under-provisioning. The absolute lowest value found was 0.00002947. The largest value less than $\alpha\varepsilon$ was 0.00003435. No two configurations in the 3585 were the same. All 3585 had no under-provisioning. The smallest quality larger than αε was 0.0001298 and had one instance of under-provisioning.

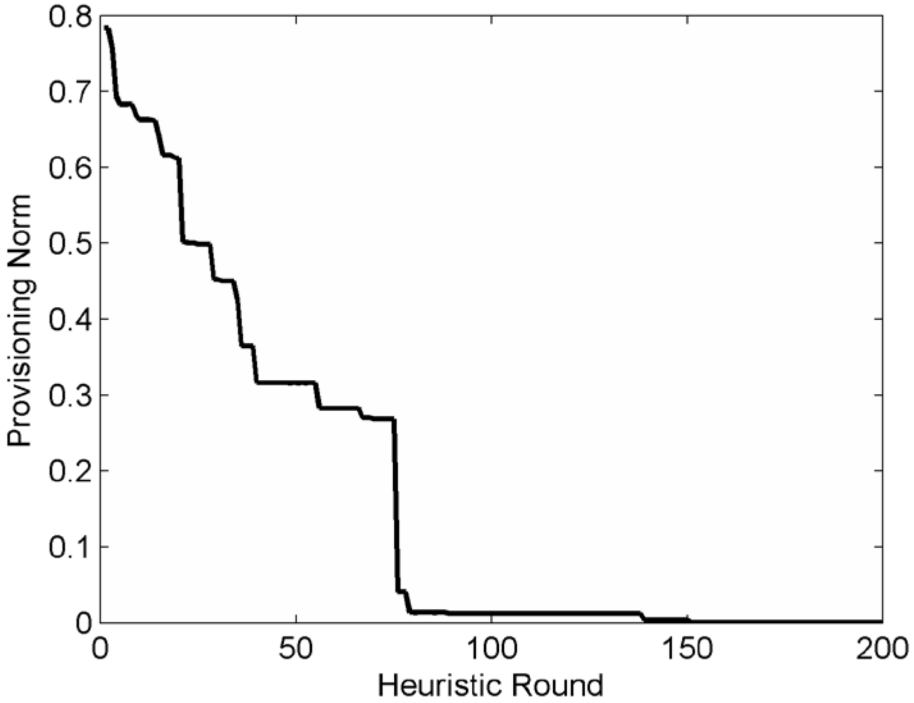


Figure 11. Example of the heuristic's decent to the best quality. Each drop is a better configuration.

In order to reduce overall execution time, subset configurations are selected from the catalog with ever increasing quality. The testbed executes each selected configuration and collects performance data. Finally, the results of the performance were related to the quality measure.

For each selected configuration, the services are assigned to their nodes. Each service is invoked by issuing http requests (the load). The number of requests per second is adjusted every second over a period of 90 seconds. The load of http requests per second per service was generated from a Poisson process (μ=10) *a priori*. Each entry in the load is a three-tuple <*time step*, *service number*, *hits per second*>. The load has 9000 entries (90 seconds • 100 services). This same load is executed against all configurations in order to ensure each service is loaded the same regardless of the configuration.

While the nodes process the requests, the performance data of the operating system, web server, and load runner is collected. Each configuration is tested for 90 seconds. Between each test, the load runners pause for 10 seconds while the next configuration is applied and the services are assigned to their new nodes.

Two hundred and fifty configurations were selected evenly from the 25,000 best configurations in the catalog of 59,767 configurations which was described in the previous section. This included all configurations whose quality was less than approximately 0.3. The configurations were ordered, least to greatest, by their quality measure (from the Provisioning Norm).

Figure 12 shows the mean response time of all services for each configuration. The x-axis is each configuration ordered by the quality. Recall that a lower value corresponds to better quality. The quality of each configuration is also plotted as the solid line to display its rise. The mean response time is plotted with points. Note the far left side where configurations have no under-provisioning and the steady response times. As the quality gets worse, so do the response times.

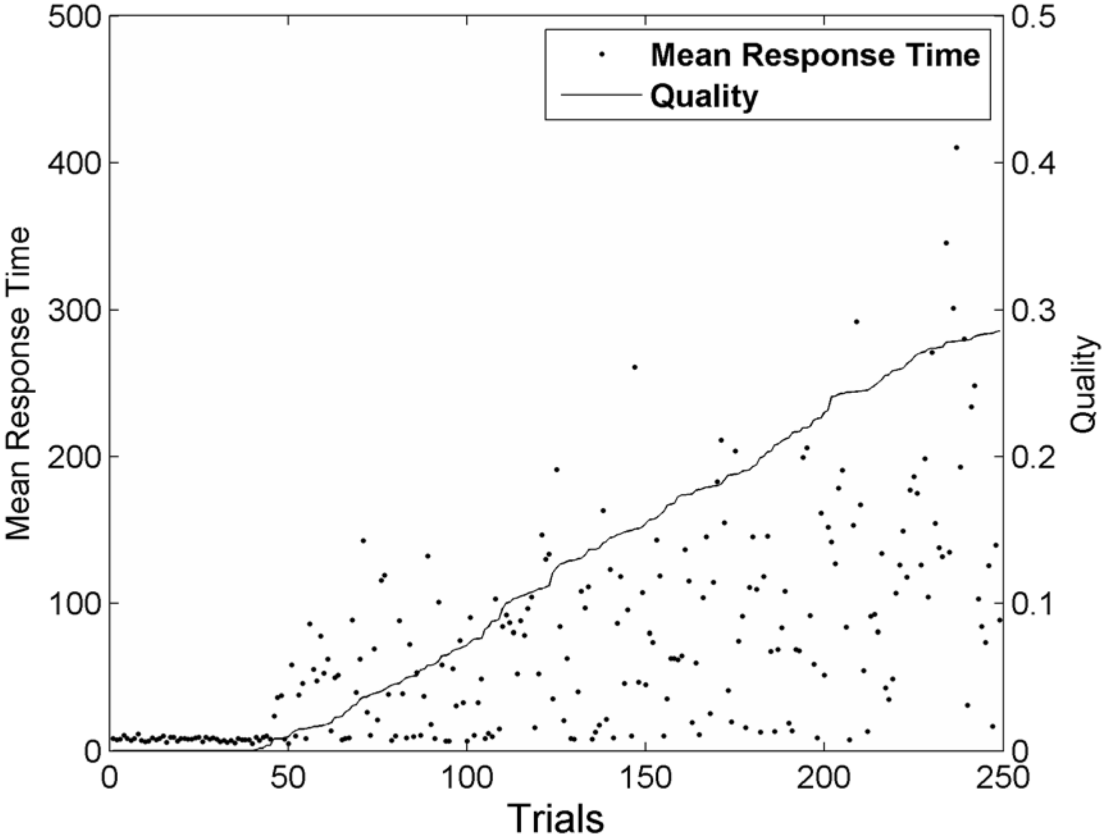


Figure 12. Performance plotted against worsening analytical quality.

The author postulates that with more accurate and more detailed profiling of services the quality measure will predict performance more accurately and tighten the correlation between the larger quality measures and their mean response time. This may include more resource types in the service and node vectors. This may also mean more than one value per resource type, e.g. maximum consumption, minimum consumption, mean consumption. This is left for future work.

Service Inclusion and Response Time

In the testbed, two experiments were performed to generate empirical results showing as α increases, services are excluded and performance improves. One experiment used a static set of nodes and sets of random services defined in Table 6. In order to make the experiment more interesting, e.g. demand closer to supply, several nodes were removed from the system. Specifically nodes 2, 7, 15, 16, and 25 were removed. This removed available resources so that the amount of demand (sum of S is 12.0581) would be closer to the supply (sum of N is 17.8200). The parameter α ranged from 0.09999 to 0.99999 step 0.1. For each iteration of α, 25 configurations of good quality were created. The 250 configurations were run, as before, for 90 seconds each while data was collected. As before, the traffic load for each service was from a Poisson distribution with λ=10.

In the second experiment, twenty-five sets of services were defined using a random process to define the CPU, memory, and bandwidth resources. CPU consumption time was generated from a uniform random distribution ranging over 500 to 900 milliseconds. Memory consumption uniformly ranged from 1500 kilobytes to 2500 kilobytes. Bandwidth consumption uniformly ranged over 5 kilobytes to 25 kilobytes. These values were selected such that the physical machine on which the virtual machines were executing would not become saturated. For each of the 25 sets of 100 services, the parameter α was first set to 0.00001 to witness services being excluded to maintain performance. In the second part, the parameter α was set to 0.99999 to witness services being included regardless of performance. In each execution, a thirty minute load is placed simultaneously on each of the 100 services. The load increases over the thirty minute trial in five minute intervals of 2, 4, 6, 8, 10, and 12 hits per second respectively. The traffic load was from a Poisson distribution generated *a priori* and used in each trial. This process is repeated for each of the 25 sets of services.

The results for the first experiment are presented in Figure 13. The results show that as α decreased more services were included and response times increased. Increasing α caused fewer services to be included, which minimized under-provisioning and resulted in significantly reduced mean response times.

On a final note, Figure 14 shows how services are not guaranteed to be included as the demand for resources grows grossly too high. This data is from 1000 random sets of 25 nodes and 100 services. For each, α is 0.00001 and the heuristic finds a good configuration. The x-axis is a ratio of the total of all service resource demands divided by the total of all node resources supplied. The number of services included is plotted as well as the quality (Provisioning Norm) of the configuration. The graph shows that as the ratio increases from near zero to 1.0 and then approaches 1.5, 100 of the 100 services are included. As the ratio of demand to supply exceeds 1.5, the number of included services decreases quickly.

Based on the theorems, the simulations, and the empirical results, α can be tuned to include more services at the cost of performance or to improve performance at the cost of inclusion. Policy drives the optimal selection of α. The policy will be driven by cost benefit analysis, contractual arrangements, legal obligations, etc. The policy will determine service inclusion and system performance.

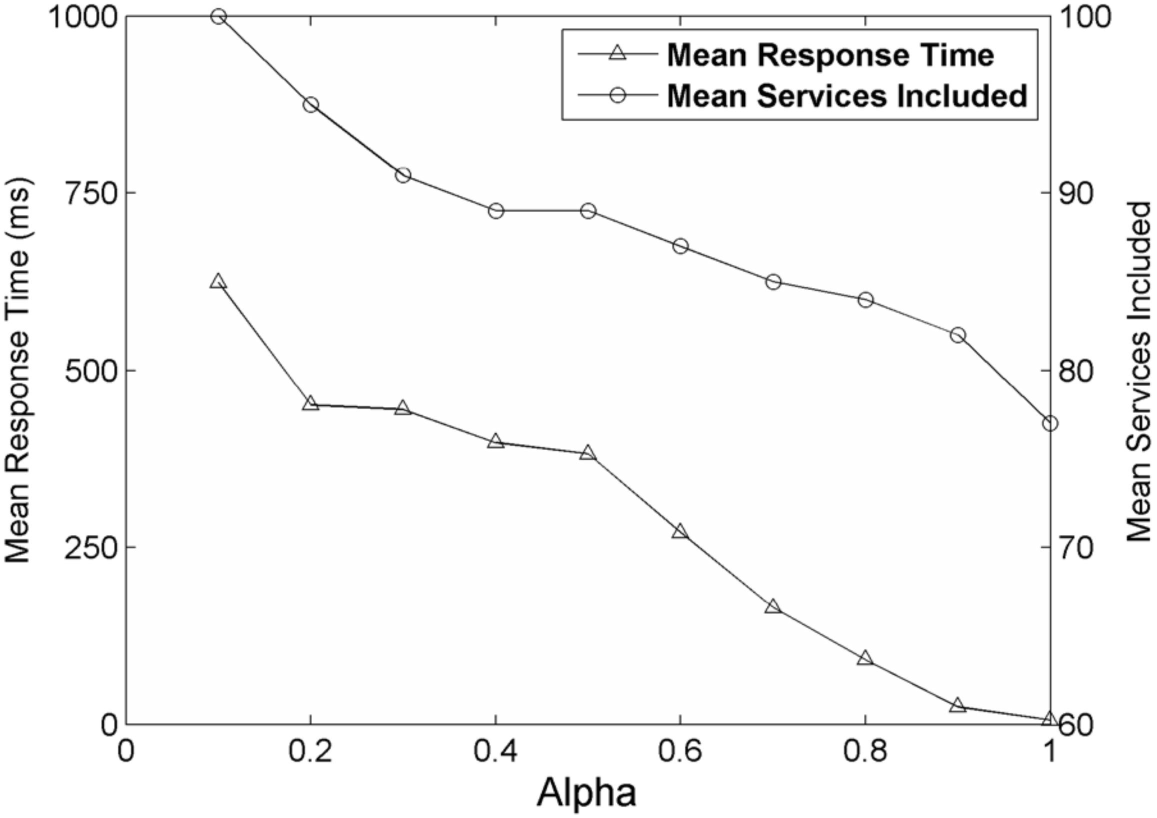


Figure 13. Empirical results using increasing values of α . Services included and Response Time plotted across α.

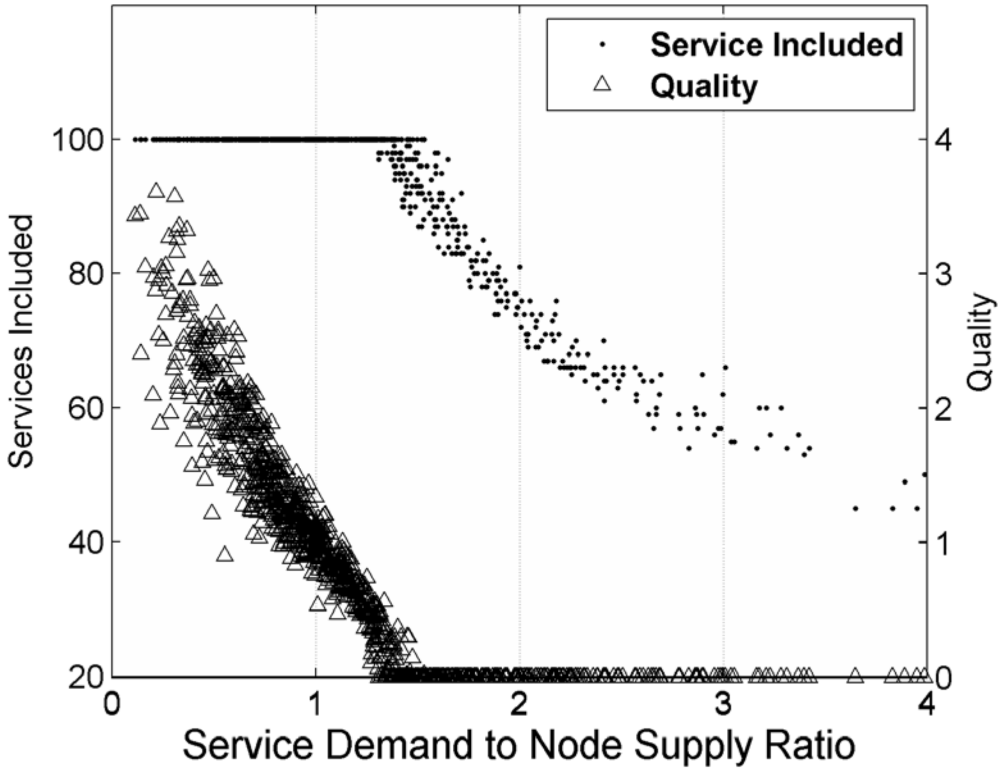


Figure 14. Services included as ratio of supply to demand increases.

The second experiment yielded similar results. Tables 7 and 3 show the results. Table 7 shows the mean response time for each case, α = 0.00001 and α = 0.99999. Each value is the average response time of all 100 services over the twenty five trials over the five minute interval. The confidence interval is 95% using the two tailed t-distribution. Clearly, α = 0.00001 maintains a low response time as the load increases relative to α = 0.99999. Note: when hits per second is 2, the dynamic profiling causes some increased response times while services are initially profiled and moved accordingly. Table 8 shows the number of services included for each five minute interval over the 25 trials.

The confidence interval is 95% using a two-tailed t-distribution. Clearly for α = 0.99999 all 100 services were always included. For α = 0.00001, services were excluded as the load increased. Because the service profile is a function of the load, the profiles grew and the Provisioning Norm mathematically excluded them.

Table 7. Mean Response Times Over Increasing Load.

Hits per Seconds	α=0.00001 Means Response Time in milliseconds (95% confidence interval)	α=0.99999 Means Response Time in milliseconds (95% confidence interval)
2	60.5 ± 1.816	88.0 ± 5.718
4	16.6 ± 0.566	883.9 ± 78.646
6	13.3 ± 0.086	1442.3 ± 115.216
8	40.6 ± 0.226	1465.2 ± 97.152
10	62.4 ± 2.523	1311.0 ± 77.370
12	80.1 ± 6.123	1227.2 ± 67.042

Table 8. Mean service included over increasing load.

Hits per Seconds	α=0.00001 Services Included (95% confidence interval)	α=0.99999 Services Included (95% confidence interval)
2	100.000 ± 0	100.000 ± 0
4	100.000 ± 0	100.000 ± 0
6	96.613 ± 0.031	100.000 ± 0
8	86.952 ± 0.116	100.000 ± 0
10	70.753 ± 0.073	100.000 ± 0
12	58.755 ± 0.061	100.000 ± 0

Other Measures

Matrix Norms

The common matrix norms are 1-norm, 2-norm, and ∞-norm. The matrix 1-norm is defined as the maximum column sum in the matrix. Alternatively, the ∞-norm is defined as the maximum row sum in the matrix. The matrix 2-norm (also called the spectral norm) is defined as the square root of the maximum eigenvalue of the product of matrix's conjugate transpose and the matrix.

Define moving from one configuration to another as changing a single entry in the adjacency matrix C from 0 to 1 or 1 to 0. This change slightly modifies the residual matrix N-CS. This modification changes the quality of the configuration for better or worse. In order to determine if it is better or worse, the before value and the after value of the matrix norm are compared. This becomes more expensive for large matrices, particularly for the spectral norm. The matrix norm must be evaluated across the entire matrix in order to determine if a local change provides improvement. Local decisions (at the node) can not be made without global consideration.

The second challenge using matrix norms as quality measures derives from the nature of the computation. The value of the 1-norm is derived from the values contained in the single maximal sum row; regardless all other values in the matrix. Therefore changes to *any other value*, unless they create a greater maximal row sum, have no effect on the quality measure provided by the norm. The ∞-norm exemplifies a similar behavior for columns instead of rows. Thus using the 1-norm and the ∞-norm as quality measures creates large equivalence classes among configurations. In other words, given a fixed sets of nodes, N, and services, S, many configurations share the same measure of quality.

The spectral norm does change when any value of the matrix changes. Therefore, any change can be considered better or worse using the spectral norm. However, this consideration only occurs by calculating the spectral norm for the entire matrix. Furthermore, the calculation is expensive relative to calculating the Frobenius norm.

Vector Norms

Vector norms can be applied to matrices by translating the matrix into a vector by appending each row into a single row vector comprised of all entries in the matrix. The 1-norm, also called the taxicab or Manhattan norm, for vectors is the sum of all absolute values in the vector. Unlike the matrix norms above, any change can be considered at the node level, i.e. if a residual resource absolute value decreases by 4 then the quality measure vector 1-norm decreases by 4. If any of the values are reduced, then this change improves the overall quality of the entire system. Local decisions of quality can be considered without global calculations.

The ∞-norm for vectors is the maximum absolute value in the vector. Like the above matrix norms, large equivalence classes are created and individual changes can not necessarily be evaluated, e.g. not all changes change the quality measure. For the ∞-norm, clearly reducing any value 'should' be better; however, the overall quality measure may not reflect a noticeable change in many cases.

The 2-norm, also called the Euclidean norm, of a vector is the square root of the sum of the squared values of the vector. This norm, compared to the 1-norm, reflects bigger changes with bigger changes to the value of norm. This is caused by the squaring of the value before summation. Larger values are magnified relative to their smaller counter parts. This increased sensitivity to change makes the 2-norm superior to the 1-

norm. The 2-norm further represents the Euclidean length of the vector. The Frobenius matrix norm is equivalent to the vector 2-norm, and is thus the best choice of norms for the application in this dissertation.

Kullback-Leibler

The Kullback-Leibler distance [64], also called relative entropy, is a distance measure between two discrete probability density functions. It is defined as

$$KL\left(p,q\right)=\sum_{k} p_k \log_2\left(\frac{p_k}{q_k}\right) .$$

As described in [7], Kullback-Leibler can be used to compare the distance between discrete resource vectors, e.g. service and node resource vectors. The best (smallest) distance is zero; implying that the available equals the demand. The better (smaller) will be where the vectors are similar, i.e. they are shaped the same.

This quality measure with a modified heuristic was used as an alternative approach to finding quality configurations. Configurations were found for random service and node matrices. Additionally, empirical performance data was collected from the testbed.

In order to properly consider measures where demanded resources exceed supplied resources and to normalize the resource values, Kullback-Leibler is modified where *s* is the demand of a service and *n* is the *available* (not yet used) resources,

$$KL^*\left(n,s\right)=\begin{cases}\displaystyle\sum_{k}\frac{n_k}{\sum_i n_i}\log_2\left(\frac{\dfrac{n_k}{\sum_i n_i}}{\dfrac{s_k}{\sum_i n_i}}\right) & where\,\forall k, n_k \geq s_k \\ \infty & where\,\exists k, n_k < s_k\end{cases}$$

In other words, $KL^{*}\left(n,s\right)$ returns the maximum value if the service does not fit on the node. This provides the numerical heuristic methods the ability to quickly avoid under-provisioning. This is similar to the implementation in [7].

Figure 15 is based on a MATLAB simulation in which 100 sets of 25 random nodes and 100 random services were generated. For each a quality configuration was found using the Provisioning Norm (α=0.99999) and the modified Kullback-Leibler. Figure 15 is the plot of how many services and nodes each method included in the 100 configurations. These are plotted along the ratio of total service demand to the total node supply.

Figure 15 shows some interesting behavior differences between the two methods. First, note Kullback-Leibler uses as few nodes as possible where Provisioning Norm uses all nodes by assigning services across nodes more evenly. This behavior manifests in the lower left part of the graph where at a low demand/supply ratio Kullback-Leibler only uses about half of the 25 nodes and slowly increases as the demand increases. Second, note at the upper left part of the graph that Provisioning Norm always includes 100 of the services at reasonable demands where Kullback-Leibler does not include 100 most of the time. Third, as the demand/supply ratio approaches and passes 1.0, both methods include a similar number of services.

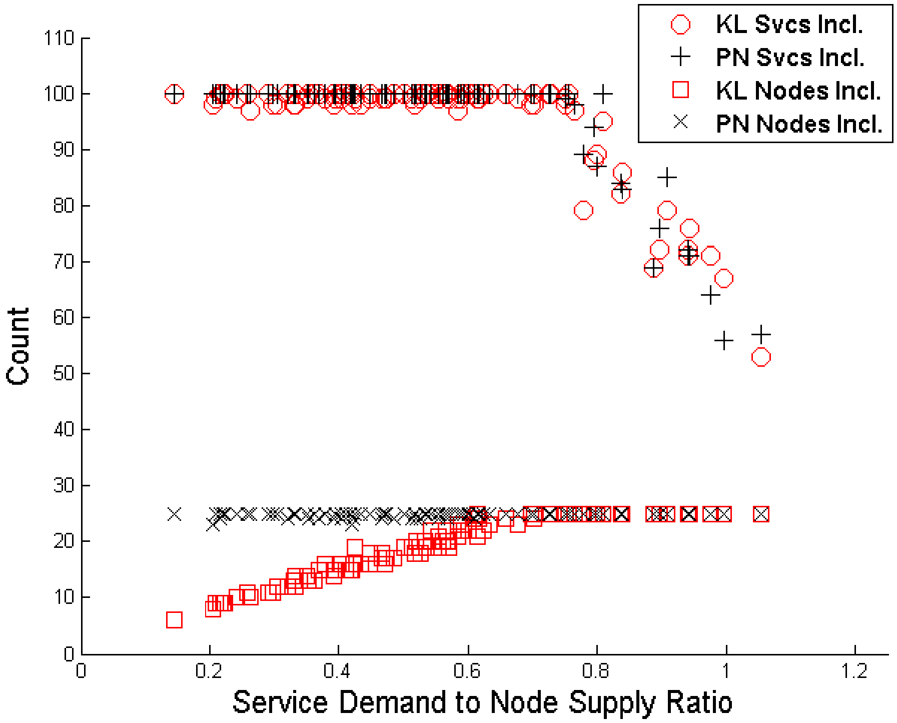


Figure 15. Kullback-Leibler and Provisioning Norm (*α*=0.99999) services and nodes included plotted across the ratio of service demand to node supply.

Heuristic

Our work utilizes a combinatorial heuristic similar to [62][63]. Resende [63] describes a multi-start, hybrid heuristic. The heuristic using a tunable parameter is defined as multi-start by its use of several points in the search space as starting points. Each start point creates inputs into each phase of the heuristic. The hybrid nature of the heuristic is the combination of several methods; each is a phase in the process. These include local search, path-relinking, elite solutions, intensification, and post-optimization. The authors show the efficiency and effectiveness of the heuristic by applying various combinations of heuristic phases: all of the phases, all except post-optimization, and all except post-optimization and path-relinking. The entire heuristic is the most effective

[63] but not as fast as the others. The heuristic is primarily based on their heuristic without path-relinking and without post-optimization. This section introduces the Iterative Configuration Method. The method is first discussed informally. Pseudo code is formally defined using set theory. Finally the method is discussed in terms of the matrices previously discussed.

The heuristic is informally defined as follows. At a high level, it is a random, greedy algorithm which removes from nodes services which are not contributing to the quality and adds to nodes services which do contribute to the quality. The heuristic is an iterative method similar to Newton’s Method, Secant Method, and Conjugate Gradient Method. In those methods, the next move toward the minimum is calculated and is otherwise not restricted to other conditions or choices. The heuristic selects the next move toward the minimum based on a set of possible moves. This set, as described below, is limited in changing the existing configuration to removing or adding a service to a node. This difference limits the walk through the solution space, where the methods mentioned above are free to calculate whatever walk is best, i.e. steepest decent, etc.

Given a set of nodes, a set of services, and a randomly selected mapping between them (a configuration), the goal is to find the mapping such that the quality of the mapping is optimized (low). Services which are not contributing to the quality are dropped. Available services which could contribute to better quality are added. A node is randomly selected. The quality is calculated for this (and only this) randomly selected node. The measure is then calculated for the node iteratively excluding each of the node’s services. If the node’s quality is better without a particular service, the configuration is changed by removing the service from the node. Next, for each service not on a node, the quality is calculated for the node as if the available service were on the node. If the node's quality is better including the available service, the configuration is changed by adding the service to the node. The process of randomly selecting node is performed a parametric number of times (*depth*) as the overall configuration's quality measure converges to a minimum. The overall process of starting with a random mapping is performed a parametrically specified number of times (*breadth*). The mapping with the overall best quality selected as the optimal.

The heuristic is defined formally in Figure 16. In Figure 16, the sets N and S are respectively a set of nodes and a set of services. From these, a set $\boldsymbol{N}$ is selected from the power set of N and $\boldsymbol{S}$ from the power set of S. $\mathbb{C}$ is the power set of all ordered pairs from $\boldsymbol{N}$ and $\boldsymbol{S}$. This power set is filtered down to $\mathbb{C}^*$, a set of sets, such that each contains only ordered pairs where a service is paired with a single node. In other words, $\mathbb{C}^*$ is a set of functions from proper subsets of $\boldsymbol{S}$ to proper subsets of $\boldsymbol{N}$. These can be thought of as possible configurations between $\boldsymbol{S}$ and $\boldsymbol{N}$. The function Q, given a configuration, set of nodes, and a set of services, returns a non-negative real value. This non-negative real value is a measure of the quality of the configuration given the set of nodes and the set of services, where 0 is perfect and lower values are better. The parameters *breadth* and *depth* define how far and deep the heuristic will look.

Continuing on in Figure 16, the heuristic takes in $\boldsymbol{N}$, $\boldsymbol{S}$, $\mathbb{C}^*$, Q, *breadth*, and *depth* as inputs and it returns the best possible configuration. The *breadth* parameter determines how many initial random configurations C_1 from $\mathbb{C}*$ will be selected. Once a C_1 has been selected, its quality and the initial configuration are captured as the starting point. The *depth* parameter determines how many times a random node, *n*, is selected for

```
Let $\mathcal{N}$ be a set of nodes.
Let $\mathcal{S}$ be a set of services.
Let $S \in \mathcal{P}(\mathcal{N})$ and $N \in \mathcal{P}(\mathcal{S})$
Let $\mathcal{C} \subset \mathcal{P}(N \times S)$ be all possible ordered pairs.
Let $\mathcal{C}_* = \{C : C \in \mathcal{C} \wedge ((x, y), (x, z) \in C \rightarrow y = z)\}$
Observe $\mathcal{C}_*$ is the set of functions from $N$ to $S$ s.t.
 $Domain(C \in \mathcal{C}_*) \subseteq N$ and $Range(C \in \mathcal{C}_*) \subseteq S$
Let $Q{:}\mathcal{C}_* \times \mathcal{N} \times \mathcal{S} \rightarrow \mathbb{R}^+$ (0 is perfect)
Let $breadth, depth \in \mathbb{N}^+$

$Heurisitc(N, S, \mathcal{C}_*, Q, breadth, depth)$
    For $i = 1$ to $breadth$
        Select a random $C_1 \in \mathcal{C}_*$
        $q_* = Q(N, S, C_1)$
        $best = C_1$
        For $k = 1$ to $depth$
            Select a random $s \in S$
            $C_s = \{(x, y) : (x, s) \in C_k\}$
            $N_s = \{x : (x, y) \in C_s\}$
            $q_s = Q(C_s, N_s, \{s\})$
            $C_k = C_k - \{(x, s) : (x, s) \in C_s \wedge$
                        $Q(C_s - (x, s), N_s, \{s\}) < q_s\}$
            $N_a = N - \{x : (x, y) \in C_k\}$
            $C_k = C_k + \{(x, s) : x \in N_a \wedge$
                        $Q(C_s + (x, s), N_s, \{s\}) < q_s\}$
        End For
        if $Q(N, S, C_k) < q_*$ then $best = C_k$
    End For
    Select $best$
```

Figure 16. Heuristic algorithm formally

improvement. In each iteration of the loop, a random n is selected from $\boldsymbol{N}$. C_s is a configuration of ordered pairs for n in the overall configuration C_k, in the k^{th} round. $\boldsymbol{S}_\mathbf{n}$ is the services assigned to n. Now, with this configuration, C_n, this set of services, $\boldsymbol{S}_\mathbf{n}$, and this node n, q_n is determined to be the present quality measure for this node n.

Next the configuration C_k will be altered by removing all services in $\boldsymbol{S}_\mathbf{n}$ such that the quality of C_k is better off without those services. The quality measure Q is calculated for the configuration C_n - (x,n). If this measure is less than q_n, then the node is better off without (x,n).

Next consider all of the available services in $\boldsymbol{S}$ that are not assigned to any node, $\boldsymbol{S}$ - $\{x:(x,n)\in C_k\}$. All available services, (x,n) added to C_n and whose quality is less than q_n are added to the overall current configuration C_k.

By selecting a random node and removing and adding services *depth* number of times. The initially random configuration is transformed in a configuration of better quality. If its quality is better than the best determined so far ($q*$), then it is selected as the best so far. This process continues for *breadth* number of times.

Having formally defined the heuristic, the heuristic is further defined in terms of the matrices presented previously. The following product of the configuration matrix with its transpose creates an adjacency matrix where the c_{ij} entry in C^* has a 1 if the i^{th} service and the j^{th} service are assigned to the same node. Note that for all $i=j$, the diagonal on C^* is all 1's.

$$C^* = C^T C \tag{9}$$

In order to remove the services from the nodes they are assigned to, the diagonal is subtracted such that the diagonal of C_{-}^{*} is 0's.

$$C_{-}^{*} = C^{T}C - diag\left(C^{T}C\right) \tag{10}$$

Utilizing a series of manipulations on *C*, a matrix is derived providing a relationship between all nodes and all the available services (not currently used for a node). In brief, a vector is calculated by summing the columns of *C*. The vector is transformed using the logical *not()* operator (e.g. zeros become ones and non-zeros become zeros). In the MATLAB implementation, this is done on a per node basis. Consider (11) a matrix that makes the unused services available for mathematical consideration for each node.

$$C_{+}^{*} \tag{11}$$

Given these three configuration matrices, the service resource matrix *S* is multiplied respectively by each configuration matrix (9), (10), and (11).

$$C^{*}S \tag{12}$$

$$C_{-}^{*}S \tag{13}$$

$$C_{+}^{*}S \tag{14}$$

The following matrix (15) provides a row for every service. The row for each service is the resource values required by the node to which it belongs. For example, a node with three services will have its row from *N* appear once for each of its three services in S^{*}.

$$N^{*} = C^{T}N \tag{15}$$

The following equation (16) defines a row wise quality measure. Given a matrix *A* of *m* rows and *n* columns, the *i*th entry in the resulting single column matrix is the quality for the *i*th row in *A*.

$$\begin{aligned} &Q:\mathbb{R}^{m\times n}\to\mathbb{R}^{m}\\ &Q_i\left(A\right)=\left\|A_i\right\|_Q \end{aligned} \qquad (16)$$

The row wise norm (16) is applied to the difference of the matrices providing the resource and the matrix of the required resources. The D^* is a single column matrix, (17), with an entry for each service. The entry is the quality of the node to which the service belongs, e.g. all of the services of a particular node will have the same value. The D_-^* has an entry for each service. The entry is the quality for the node *excluding this particular service*. The D_+^* has an entry for each service. The entry is the quality for the node *including the particular service*.

$$D^*=Q\left(C^*S-N^*\right) \qquad (17)$$

$$D_-^*=Q\left(C_-^*S-N^*\right) \qquad (18)$$

$$D_+^*=Q\left(C_+^*S-N^*\right) \qquad (19)$$

The difference between the current configuration's quality (17) and the configuration without services (18) is a single column matrix. The row that has the highest, positive value represents the service that should be removed.

$$D^*-D_-^* \qquad (20)$$

Similarly, the highest, positive value of the difference between the current configuration's quality (17) and the configuration with new services (19) determines the service that should be added.

$$D^* - D_+^* \tag{21}$$

In summary, using a quality measure (objective function), services are candidates to be removed and added; greedily using some simple matrix manipulations. This approach is a greedy or opportunistic heuristic. In order to reduce, the potential detrimental effects of greed (e.g. deterministically settle on a bad local minimum), randomization is used in the process as described in Figure 16 above.

Related Works

Our work is most generally defined as an optimization constraint problem, a multi-dimensional, multi-choice knapsack problem [39], and a specialization of the Application Placement Problem [6]. The work most related to the work presented here is [7]. In [7], Zhang et al. describe the On-Line Tenant Placement Problem. It differs from the Application Placement Problem and the Online Application Placement Problem [39] because applications do not require containers, e.g. these are executable code. Tenants in SaaS reside in a service container such as a web server, database server, or middleware server. Zhang et al. offer a heuristic, the Tenant Dispatch Heuristic to address the On-Line Tenant Placement Problem. Zhang et al. further consider a multi-resource environment where each tenant requires multiple types of resources such as processor utilization, memory consumption, and bandwidth. The Tenant Dispatch Heuristic maps arriving services to existing nodes using Kullback-Leibler Distance. As the Kullback-

Leibler Distance between the service and the node decreases, the matching of resource demand of the service to the resource availability of the existing node improves. Once a service is placed on a node, it is not moved. Using two dynamically tunable parameters, nodes can be systematically added in order to provide resources to growing demand. Unlike our work, they assume that tenants (services) cannot be moved once assigned. They further assume that more servers (nodes) can be added as needed to meet demand, i.e. no under-provisioning is allowed.

Speitkamp et al. [8] discuss a different but isomorphic problem. The authors explore server virtualization by analyzing workload and optimally assigning its virtual machine to a physical host. Our work assigns services to nodes, physical or virtual, where they assign virtual machines to servers. A significant difference is the consumption assigned to the physical nodes includes operating system overhead. The assignment presented here is independent of the virtual or physical nature of the compute resource. The testbed presented in our work is virtualized in order to increase the number of available hosts given the limited physical resources. The authors used linear programming with some relaxation to find acceptable configurations. Their model supports multi-dimensional resources, but their experiments considered only CPU utilization. Our model, testbed, and experiments implement three resources: CPU, memory, and bandwidth.

Urgaonkar et al. [6] describe applications as having multi-tier capsules. They demonstrate that Application Placement Problem is NP-Hard. In the offline condition/environment/scenario Application Placement Problem is addressed as a matching problem on a bipartite graph with LP-relaxation. Online Application Placement

Problem is shown to reduce the placement problem to the minimum-weight perfect matching problem. The application capsules and servers are described using only a single resource value.

Yu et al. [65] offer two models and several algorithms for service composition under the multiple quality of service constraints for services with sequential or general flow structures. Their application of heuristics to a multidimensional multi-constraint 0-1 knapsack problem model is similar to our work. They similarly point out optimal algorithms such as branch-and-bound techniques take too long to execute in a runtime environment. They show efficient heuristics are, while less than optimal, a sufficient solution for a runtime environment of service composition.

Ricci et al. [66] describe the network testbed mapping problem. The network testbed consists of compute nodes, routers, switches, links, etc. A test scenario needs a set of these resources. Ricci et al. focus on a solution to quickly select a mapping of requirements to resources in an NP-Hard problem. Their solution is based on simulated annealing. Resources are not shared at run time. Each scenario is executed one at a time, i.e. not a multi-user environment.

Karve et al. [4] discuss maximizing the satisfied demand of services while limiting the interference of node introduction. Their approach also balances resource usage across nodes and minimizes configuration changes. The authors use a model of two resources: CPU utilization (load-dependent) and memory (load-independent).

Several other works are similar in various natures and approaches. The work of Zhu et al. [10] and Hyser [67] et al. are similar to this work. In [10], an automated capacity and workload system is described to manage the assignment of virtual machine instances to physical machines. The system provides management in a three tier hierarchy: the local node, the collection of nodes, and the collection of collections of nodes. They use the simulated annealing method described in [67] to obtain a near optimal configuration that determines which physical machine should host which virtual machines.

Tang et al. [68] present a peer-to-peer grid computing model to maximize resource utilization. In particular, they describe each node's resources using a matrix of attributes where each job seeks the node that best meets its needs. Each node keeps resource information about itself and others. The search is relayed from node to node until a match is found and returned.

Wang et al. [69] developed a resource management framework for multi-tier web applications. The multiple, multi-tier applications framework is similar to our multi-dimensional resource service matrix. They similarly define the problem as a non-linear constraint problem but measure performance using deterministic and stochastic methods.

Steinder et al. [5] describe using a job/web work profiler which determines the number of CPU cycles required to execute the workload. These values for all of the workloads, like the service resource vector in our work, determine the virtual machine configuration in a large data center.

In other autonomic service oriented architecture works, [45], [46], [9], the focus is based on maximizing profits while maintaining multiple tier service level agreements. Almeida et al. in [45] break the resource allocation problem into a short term arrangement problem and a long term allocation problem. They utilize queuing and optimization techniques in their model and performance measurements. Ardagna et al. in

[46] similarly model and measure with queuing and optimization techniques to assign virtual machines to CPUs in a physical server. A self-adaptive capacity management framework described in [9] uses queuing and optimization in a multi-tier virtualized environment to demonstrate significant profit gains relative to a static model.

Summary

This chapter has presented the Provisioning Norm, a repeatable parameter-driven method, to assign web services with resource demands to servers with resources. This matching between services and servers is based on both performance and inclusion considerations. The Provisioning Norm features the ability to find configurations across the entire spectrum of possible performance and inclusion policies. This spectrum ranges from *guaranteed performance* to a maximal number of services to guaranteed maximal performance to *all services*. This is critical for the practical implementation of a Platform as a Service or Software as a Service as it provides enormous flexibility to the system's administrators.

Empirical testbed results show great promise for the resource provisioning in an environment that is closer to real-world circumstances than was the case in a large body of related work. Our testbed included multiple resource types. Few pieces of related work considered multiple resource types in their models; even fewer considered more than the single resource type of CPU utilization in their empirical tests. Considering multiple resource types is a critical aspect of the service placement problem because Web Services are complex, occurring in many shapes and sizes. Moving beyond CPU utilization allows for enhanced resolution in service resource consumption leading directly to enhanced service placement. The asymmetric norm developed and the testing performed in this overcame the challenges presented by multiple resources types: the model is more complex, calculations take more time, and pragmatic implementation is difficult.

The methods developed in this dissertation show great promise for managing large-scale service environments. These environments can be regionalized based on geographic or logical divisions such that each implements a different policy. The Provisioning Norm lends itself to parallelization in regards to its component matrix and heuristic operations allowing large scale results in smaller time frames.

V. On-Line Service Profiling, Self-Organization, and Caching Configurations

Introduction

Software as a Service (SaaS) is the name given to the cloud computing model in which a vendor provides to the consumer access to software running on the vendor’s hardware. This relationship allows the consumer to utilize all the benefits of the software without the cost and challenge of administering the hardware and software themselves. This also provides the vendor with an economy of scale.

This work specifically discusses web services as tenants of a web server. The work can be extrapolated to include database services and other middleware services where the customer executes software (services) as request and response transactions. These tenants are scripts, executables, or objects requested from the outside which execute in a server side process and return data. The server side process is a web server, or even a database server.

As the technology improves, as the economies of scale are realized, and as the administrative freedoms increase, cloud based computing models, such as SaaS, have the potential to become the dominant mechanism of providing services to end users. To manage this growth the providers of SaaS need increased autonomicity of these systems; the systems must self-manage. The large scale environments of the future will out pace the abilities of individual administrators to monitor their performance and implement system level policies.

The Cloud Chamber, introduced in this work, and described in Chapter 3, is a testing environment where services are modeled and executed, and autonomic policies and methods are tested. We believe emulating services in real life web server environments is key to understanding how to model and measure the behavior of services in the wild.

As described in Chapter 3, our requirements for the Cloud Chamber were generated from this approach. They include but are not limited to the following. The ability to test heterogeneous services on heterogeneous servers is critical. Services requiring different amounts of computing resources execute differently on servers providing different amounts of computing resources. The environment is limited to a virtualized environment such as VMWare, Xen, etc. Performance data is collected in real-time to a SQL database. The traffic of requests to the services can be generated based on a pre-recorded load of arbitrary length. The assignment of services to servers are autonomic based on interchangeable algorithms and numeric methods which react to changes in the environment such as increased traffic load, loss of a server, or policy changes.

The Online Tenant Placement Problem describes the arrival of web services and their placement on web servers. In this work we propose a generalization of the problem. In addition to service arrival, services change their resource demands and can be moved from one server to another. The Cloud Chamber is used to examine the generalized form of the Online Tenant Placement Problem. This generalized form allows for the placed tenants to change in size, thus requiring them to potentially be moved. This work specifically examines the effects of caching service placements on performance, quality, and cost of change. Finding a particular placement configuration is one aspect of the problem. However, the critical aspect of the Online Tenant Placement Problem is how the system as a whole behaves as it transitions from one configuration to the next under ever changing conditions. In this work, changing conditions are defined as changes in the requests a service receives per second. Because the Cloud Chamber moves beyond simulation into real-time service execution, performance data such as response times and http codes can be examined. Furthermore, the subtle interactions of multiple services manifest on real web servers in a manner that is not observable in the simulated environment.

In an attempt to address this matter, this work measures over time the performance of the services, the number of service movements during reconfiguration, and the quality of discovered configurations. These measurements are taken across multiple trials which implement different caching schemes. We found that implementing a caching scheme using the current configuration and a single random configuration as the only two seeds for the search heuristic maximized performance with minimal cost of change. Using caching with no random seeds minimizes the cost of change but with degradation in performance as measured by an increase in response times.

The next section presents related and foundational works by others regarding testbed environments and the mechanisms employed in Cloud Chamber. The third section presents how the services and servers are mathematically modeled, measured, and finally organized into profiles. The fourth section outlines the self-organizing, autonomic aspects of the Cloud Chamber such as its gossip network and heuristic search methods. The fifth section provides the analytical foundation for the expected caching behavior. The work finishes with an experimental setup and results demonstrating the results of caching.

Related Works

Urgaonkar et al. [6] describe applications as having multi-tier capsules. They demonstrate that Application Placement Problem is NP-Hard. In the offline condition/environment/scenario Application Placement Problem is addressed as a matching problem on a bipartite graph with LP-relaxation. Online Application Placement Problem is shown to reduce the placement problem to the minimum-weight perfect matching problem. The application capsules and servers are described using only a single resource value.

Juszczyk et al., outlines the Genesis2 [70] and CAGE [71], the most similar works to our testbed. Genesis2 is a framework in which varieties of SOA testbeds can be created. It allows researchers to model web services from which it creates an executable testbed. Genesis2 provides generation of traffic, monitors performance, and allows for plug-ins and modification on the fly. As of this writing, it is not yet available for download. Its primary difference with our work lies in its generalization. Our testbed is specifically interested in testing on-line self-organizing methods.

Bianculli et al. [72] describe SOABench as a framework for the automatic generation and execution of testbeds for benchmarking middleware in SOA architecture. In this work SOABench tests three middleware servers, finding their weak points under heavy loads. This framework differs from ours, but not significantly, due to its focus on middleware. They are not focused on placing or moving services in reacting to performance but are more focused on investigating static performance in order to find weaknesses.

Bertolino et al. [73] offer PUPPET as a means to test web services under development that rely on unavailable external web services. The external services are unavailable for testing due to various reasons such as implications to production, cost of usage, or other side effects. PUPPET creates a stub for the otherwise absent services that performs per its prescribed functional and service level agreements.

Several other works implementing a real world web service testbed deploy only a single service. Although these are multi-tiered or composite services, they differ from ours in that ours is a collection of heterogeneous services on heterogeneous servers. Two quality, representative examples of this are Liu et al. [74] and Iqbal et al. [75]. In [74] they deploy TPC-W [76] a service commonly used for benchmarking. In [75] they deploy RUBiS [77] an open source web based auction application commonly used for benchmarking.

In other autonomic service oriented architecture works, [9], [45], [46], the focus is on maximizing profits while maintaining multiple tier service level agreements. Almeida et al. in [45] break the resource allocation problem into a short term arrangement problem and a long term allocation problem. They utilize queuing and optimization techniques in their model and performance measurements. Ardagna et al. in [46] similarly model and measure with queuing and optimization techniques to assign virtual machines to CPUs in a physical server. A self-adaptive capacity management framework described in [9] uses queuing and optimization in a multi-tier virtualized environment to demonstrate significant profit gains relative to a static model.

More recently, Totok et al. [78] create service resource usage profiles based on service access attributes across the tiers and place sensors in the client, middleware, and database. TPC-W requests are profiled *a priori* into a Markovian model. They use their request profiling infrastructure to address four problems, showing improvement in each.

As discussed in the self-organization section, the testbed currently utilizes a combinatorial heuristic based on [63] to quickly find quality configurations measured by the Provisioning Norm [15]. Resende [63] describes a multi-start, hybrid heuristic. The heuristic, using a tunable parameter, is defined as multi-start by its use of several points in the search space as starting points. Each start point creates inputs into each phase of the heuristic. The hybrid nature of the heuristic is the combination of several methods; each is a phase in the process. These include local search, path-relinking, elite solutions, intensification, and post-optimization.

Kempe et al. [79] provide an analysis of simple gossip-based protocols. In particular they describe the information dynamics of these protocols and the effective fault-tolerance and self-stabilization of the population as a whole. They investigate mathematically the probability of diffusion using Uniform Gossip, which can be described as a simple push gossip protocol implemented in this work as a component of our hybridized gossip network.

In [80], Jenkins et al. describe a gossip-based multicast protocol implemented in an environment of multiple process groups. Their modified push gossip protocol, referred to as Gravitational Gossip, provides a mechanism for trading off the quality of information updates to reduce the amount of overhead required to deliver messages. Whereas in their work the participating nodes are characterized by values of infectivity and susceptibility to determine the required quality of information updates, in our work we characterize the messages themselves to determine the required quality of information updates and from that the mechanisms by which they are reconciled into views, viz. instantaneous node messages are all equally infectious, but the best quality configuration is required to be highly infectious.

Profiles, Models, and Measures

Services require specific computational resources to function. These resources include, but are not limited to, CPU cycles, short term memory, long term storage, and bandwidth. As discussed in [6], these requirements can be seen as a type of workload profile for the service. The profile is a vector of numeric values indicating the service's resource requirements. Each element of the vector represents the required amount of a type of resource. Each entry is normalized to the interval [0,1]. In the Cloud Chamber services have three categories of resource requirements: CPU, memory, and bandwidth; a service is described as a three-valued vector, <0.02 0.10 0.04>. These vectors are referred to as profiles throughout this dissertation.

In the Cloud Chamber, a service is an executable program on an individual virtual web server (a node). One node can service many services. A more complex, hierarchical service (N-tier) requiring multiple computational nodes is decomposed into its separate (albeit dependent) constituent services. Each is treated as an individual.

The services' resource requirements are fulfilled by the physical or virtual computational nodes upon which the services are executed. Similar to a service, a node is modeled as a vector of resources. The node's resource vector indicates the resources provided by the node to the system. The number of elements in the node vector is equal to the number of elements in the service vector; each corresponding to CPU, memory, and bandwidth.

To reiterate, the profile is a numeric representation of resources. A node's profile is based on its resource capacities. A service's profile is based on its resource consumption. The profile results from the quantification of resource measurements into a range of normalized values. The profiling performed in the paper assumes that the different resources are to be weighted equally. In order to accomplish this, the normalized value of each resource belongs to [0,1]. For the nodes, the CPU MHz ranges up to 1000 MHz, the memory ranges up to 512 MB, and bandwidth ranges up to 100 Mbps. A node with resources 1000 MHz, 512 MB, and 100 Mbps is profiled as <1.0, 1.0, 1.0>. A node with resources 500 MHz, 128 MB, and 1 Mbps is profiled as <0.5, 0.25, 0.01>.

Each virtual machine's virtual hardware constraints are implemented in VMWare. Memory size is defined when the virtual machine is created. CPU and network card speeds are not yet definable in the virtual machine definitions, so alternate mechanisms are used to enforce them. CPU speed is defined in Resource Allocation. In VMWare ESXi 4.0, individual port groups are assigned to each virtual machine. A port group's traffic shaping settings allow throttling bandwidth. Four levels of each resource type are defined for the testbed. CPU levels are 1000MHz, 800MHz, 600MHZ, and 400MHz. Memory levels are 512MB, 256MB, and 128MB. Bandwidth levels are 100Mbps, 10Mbps, and 1Mbps. This allows for 36 different sized nodes. A 800MHz, 256MB, 10Mbps node is profiled as <0.8 0.5 0.1>. As the technology exists, the node's resources can change on the fly. The bandwidth limiting values can change without rebooting the node because this is a function of VMWare. Similarly the limits on CPU speed can be changed on the fly. Memory changes only take effect after the node has been rebooted. For this paper node resources are considered fixed.

Services are similarly profiled on the same relative scale. The Cloud Chamber dynamically profiles services. This is a four step process: data collection, data aggregation, best fit linear model, and finally normalization to the aforementioned normalized scale. Per second consumption of each of the three resource types is logged into a revolving Resource Usage Window. The windows are an array with an element for each second. The window is 20 minutes long and wraps around as needed. Similarly, the services' requests (hits) per second are recorded into a Hits Window. Periodically, the Resource Histogram and Hits Histogram are built from the data in their respective windows. The Hits Histogram indicates the various rates of traffic experienced by the service. The Resource Histogram is the per hit mean of resource consumption at various rates of traffic experienced by the service. For example, in the nth position in the Hits Window, the rate is 14 hits per second. The corresponding values in the nth Resource Usage Window position are accumulated in the Resource Histogram at the 14th position. Once built, the Resource Histogram is a scatter plot of the service's per hit resource consumption at various levels of hits per second. Using Box-Muller [55], a linear model is developed for each resource type. This linear model takes hits per second as an input and yields the corresponding expected resource consumption. The hits per second are currently determined using an average over the past 60 seconds. More advanced methods for tracking hits per second can be incorporated in the future. This mean hits per second is the input to the linear function yielding a value approximating the total amount of expected resource usage. This linearly approximated value is normalized to the interval

[0,1] by dividing it by the maximum resource value, such as 512MB or 100Mbps. These three values, CPU, memory, and bandwidth are the service's profile. The profile is fed to the *nodeman* process for self-organization, described in the next section.

Self-Organization

The *nodeman* process implements a gossip messaging protocol between nodes to (1) maintain the view of nodes and services within the population and (2) generate a configuration based on these views. The purpose of the gossip network is to develop an environment of nodes which autonomically discovers the population of both nodes and services and autonomically calculates a configuration placing services on nodes.

Nodes send messages in rounds to propagate information contained within their views. A view is defined as a set of data on which the population of nodes needs to agree. Nodes maintain separate views for nodes, services, epochs, and configurations of service placement. During each messaging round, a source node selects an element from a view, a fact, and sends that fact to two target nodes. Target node selection is based on two mechanisms of population organization. The nodes self-assemble into an ordered ring topology in which each node is aware of the nodes adjacent to it. To achieve a balance between robustness and predictability under changing conditions in the environment, node selection combines the deterministic path of message propagation around the ordered ring with non-deterministic random target node selection. Every node participates in sending messages during every messaging round, and target nodes are selected both at random and deterministically by ring order. By this combination of messaging strategies every node is guaranteed to receive at least one set of facts during each messaging round. This method of selecting an element from a view and sending it along both a deterministic path and a non-deterministic path is best characterized as a simple push gossip protocol [79] hybridized by structured and unstructured network topologies [81].

During node initialization, each node joins the population and becomes self-aware through a UDP broadcast message. A node continues to broadcast its existence in each messaging round until it is both self-aware and not alone. As the population grows, the set of nodes is ordered by their unique keys to generate the structured ring topology for deterministic neighbor selection. Once the node view has become aware of a population, services are introduced through the interface via the *serviceman* process. An initial population of nodes with an awareness of services enters into the first of three population time periods. The behavior of nodes cycles through these three distinct periods referred to collectively as an epoch and individually as search, reconciliation, and settle.

The search period is a process in which nodes and services are both discovered and updated. Each node infects other nodes with elements from its node and service views, new nodes and services are discovered by the population, and nodes and services expire from the population. This allows the population and service placement to autonomically adapt to changing network environments. The node and service views are instantaneous views which are persistent across all period transitions, i.e. they are not dependent upon a specific epoch. If the nodes have a reconciled view of nodes and services, i.e. a view which is unique to a given epoch, each node independently generates a configuration using the heuristic defined in Chapter 4 and calculates the quality measure of each using the Provisioning Norm.

One or more nodes initiate a transition into the reconciliation period. During this period, the population reconciles the instantaneous node and service views into static epoch views which are used in the calculation of configurations. If configurations were generated during the previous search period, each node also sends messages regarding the best configuration it has calculated, or the best configuration that it has discovered from another node. These configurations are validated and compared by quality measure, resulting in a best configuration which is highly infectious to the population through iterative messaging rounds [81].

A third period, the settle period, follows reconciliation. This settle period accommodates variations in node awareness of the timing of the period transitions. The relative end time of the reconciliation period is synchronized through a timing message which is pushed throughout the reconciliation period. Nodes in the settle period are immune from infection by messages initiating period transitions. This ensures that all nodes have completed the reconciliation process and prevents epoch churn initiated by errant nodes which have joined the population between time synchronizations.

Several tunable parameters were externalized to allow administrative adjustment of the population behavior. The frequency of messaging rounds, the duration of the search period, the duration of the reconciliation period, the duration of the settle period, and a Time-To-Live (TTL) value for individual nodes are set to satisfy the requirements of timely message propagation while limiting network traffic and computational overhead. Two parameters for the calculation of new configurations by the heuristic are also set externally. These are the number of random configurations to use as seeds and the number of cached configurations to use as seeds.

The need for parallel computation of service to node configurations required consensus of node and service views across all nodes in the population. Consensus ensures the validation of a calculated configuration and permits comparison and selection of a best configuration. The implemented gossip protocol is a hybrid of a probabilistic messaging system [80] and a deterministic messaging system [81]. Although this mechanism proved effective for the general dissemination of information, an adaptation was required to satisfy the need for immediate population consensus. The solution was to implement a reconciliation phase that reduces the node and service views to the minimum set of elements shared by all nodes within the population.

Note that the set of nodes and services populating the epoch view is the set reconciled from the previous search period. The persistent node and service views may be different than the set of nodes and services in an epoch view. Each epoch retains its own view of nodes and services upon which the configuration generated during that epoch is dependent. A node may have up to three other distinct views of nodes and services in addition to its persistent views. It may also have the view that will participate during the next epoch and which is currently being reconciled, the view that corresponds with the configuration calculated during the previous epoch, and the view which is associated with the configuration that was implemented during the previous epoch. Table 9 illustrates the relationship between the view of nodes and services and their presence during epoch periods.

Table 9. Epoch Reconciliation & Configuration Over TIme. E – Epoch, ER – Epoch Reconciliation

Set of Nodes & Services	E0	ER0	E1	ER1	E2	ER2	E3	ER3
Current View	E0	ER0	E1	ER1	E2	ER2	E3	ER3
Configuration In Process	-	-	ER0	-	ER1	-	ER2	-
Configuration Implemented	-	-	-	-	ER0	ER0	ER1	ER1

To ensure that service placement configurations are based on the same set of elements and a comparable best configuration is selected, a validation process of node and service counts and unique key values of each view is used to ensure that the configurations being compared are defined by identical node and service views. Once a configuration received from a source node has been validated by a target node, the source and target configurations are compared by their calculated quality measures. The better of the two is retained and propagated during subsequent message rounds.

The hybrid gossip protocol has proven sufficient for maintaining views across the population of nodes. The tunable parameters used to adjust the duration of periods, the frequency of messages, and the source of heuristic seeds permit optimization of consensus probability and configuration generation under varying test conditions. The adaptation of this protocol by the implementation of the reconciliation phase and configuration validation checkpoints provides a means of using this probabilistic messaging system to meet the requirements of data consensus.

A Simple Experiment

The experiment presented here demonstrates the self-organizational properties of the Cloud Chamber. The experiment compares the performance of two scenarios measured in mean response time. The first scenario tests services in the Cloud Chamber without autonomic reconfigurations. The second scenario tests services in the Cloud Chamber with autonomic reconfigurations. Ten nodes are available for providing resources to services. The nodes' specific profiles are described in Table 10. Each scenario started with the same thirty services, described in Table 11. Each test was run for 25 minutes and repeated 20 times.

Table 10. Actual and Normalized Node Resource Profile Values (10 of the 25)

node	cpu (MHz)		memory (MB)		bandwidth (Mbs)	
	actual	profile	actual	profile	actual	profile
1	1000	1.00	512	1.00	100	1.00
2	1000	1.00	128	0.25	1	0.01
3	400	0.40	512	1.00	1	0.01
4	400	0.40	128	0.25	100	1.00
5	600	0.60	256	0.50	10	0.10
6	600	0.60	256	0.50	10	0.10
7	800	0.80	512	1.00	100	1.00
8	800	0.80	128	0.25	1	0.01
9	400	0.40	512	1.00	1	0.01
10	400	0.40	128	0.25	100	1.00

Each trial of the test used the same service load values. This load is plotted over time in Figure 17. Five of the thirty services start with a high amount of traffic for the first ten minutes, decrease over the next five minutes, and finally reach a low traffic rate for the final ten minutes. Five other services behave in an opposite fashion going from low to high over the same 25 minutes. Ten other services of medium size are called at a steady, medium rate for the entire 25 minutes. The last ten services are relatively large in consumption but are set at a steady, low level of traffic for the entire execution.

Table 11. Actual and Normalized Service Resource Profile Values (25 of the 100).

svc	distri-bution	cpu		memory		bandwidth	
		mean or lower	stdev or upper	mean or lower	stdev or upper	mean or lower	stdev or upper
1	standard	100	10	300	20	5	1
2	standard	200	10	50	10	5	1
3	standard	300	10	250	10	5	1
4	standard	400	10	50	10	5	1
5	standard	100	10	200	30	5	1
6	standard	100	10	300	20	5	1
7	standard	200	10	50	10	5	1
8	standard	300	10	250	10	5	1
9	standard	400	10	50	10	5	1
10	standard	100	10	200	30	5	1
11	standard	100	10	50	10	5	1
12	standard	100	10	50	10	3	0.2
13	standard	100	10	50	10	3	0.4
14	standard	100	10	50	10	3	0.5
15	standard	100	10	50	10	2	0.1
16	standard	100	10	50	10	3	0.3
17	standard	100	10	50	10	2	0.11
18	standard	100	10	50	10	3	0.1
19	standard	100	10	50	10	3	0.5
20	standard	100	10	50	10	4	1
21	uniform	100	500	400	500	1	4
22	uniform	100	600	400	500	1	3
23	uniform	100	300	50	100	1	3
24	uniform	100	400	100	200	2	3
25	uniform	100	400	200	300	1	5

Initially the services' profile values are all zeros because there has been no traffic on which to calculate the profile. Therefore as the services are placed into the Cloud Chamber, the first configuration before the loads have been applied is based on profiles of zero. The time required to generate sufficient traffic to calculate an accurate profile and for that profile to propagate throughout the population and be included in the calculation of a configuration is approximately four minutes. During this bootstrapping timeframe the services do not perform well because they are not of good quality. In the first scenario (without continuous autonomic reconfiguration), reconfigurations are allowed only for the first four minutes of the first trial. This allows for a good configuration to be found prior to the inhibition of reconfigurations throughout subsequent trials of this scenario.

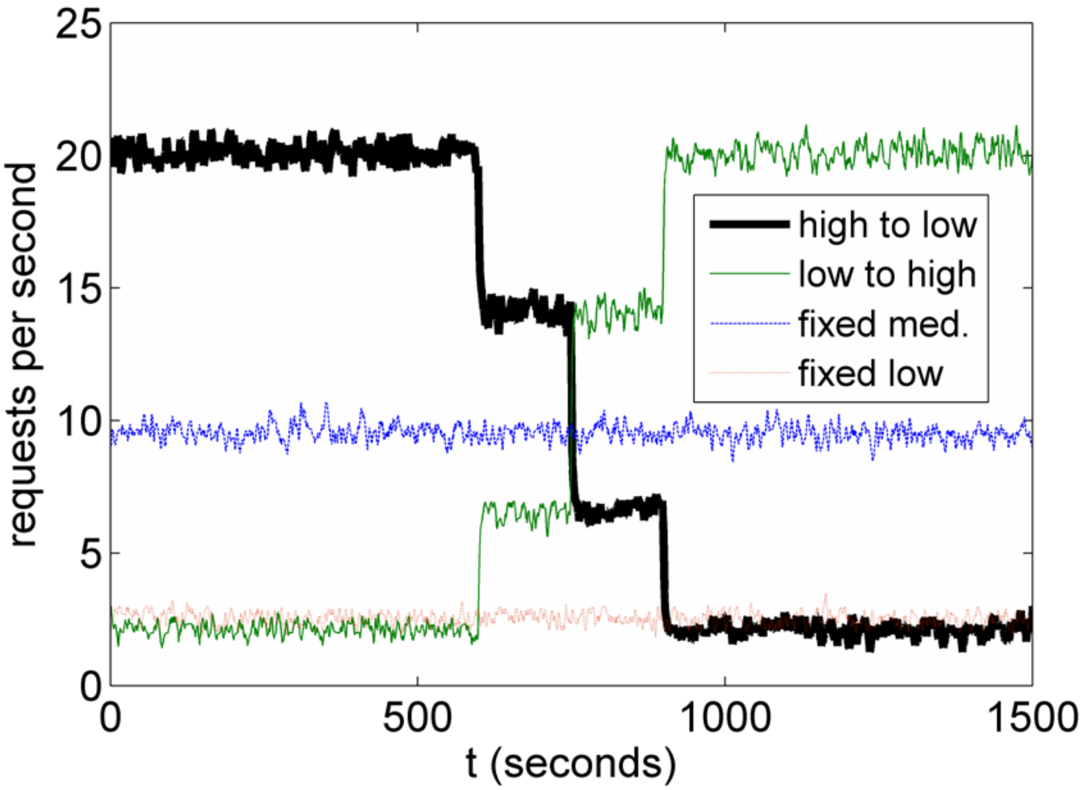


Figure 17. Predefined Traffic for Simple Experiment. Average hits per second over time for four groups of services. One group goes from low to high, another high to low, while two others are medium and low respectively.

The hypothesized results are as follows. As the traffic load of some services decreases while the traffic load of other services increases, service profiles will respectively change. As these calculated profiles shrink and grow, the scenario with reconfigurations will outperform the scenario without reconfigurations.

Results for the Simple Experiment

The results, as expected, demonstrate that reconfiguring service placement provides better performance as conditions change. Figure 18 shows the mean response times of both scenarios over the 1500 seconds. Per Figure 17, the traffic changes at t=600, t=750, and t=900. The center of the chart shows that at time t=750 and t=900 significant disruption of response time occurs. From the scenario without reconfigurations, the response time increases from less than 10ms to 500ms, and never returns. This is in contrast to the scenario with reconfigurations where disruptions occur, but are not as high and return to near previous overall performance.

Also of interest is the exceptional performance of the scenario without reconfigurations during the first 600 seconds. This is due to the fixed configuration that was calculated in the bootstrap timeframe of the initial trial for the level of traffic occurring during the first 600 seconds. The initial 240 second hump of the scenario with reconfigurations is the bootstrapping timeframe of each trial where a quality profile is not found until t=240. At t=750 and t=900, the system with reconfigurations shows increased response times but then recovers quickly.

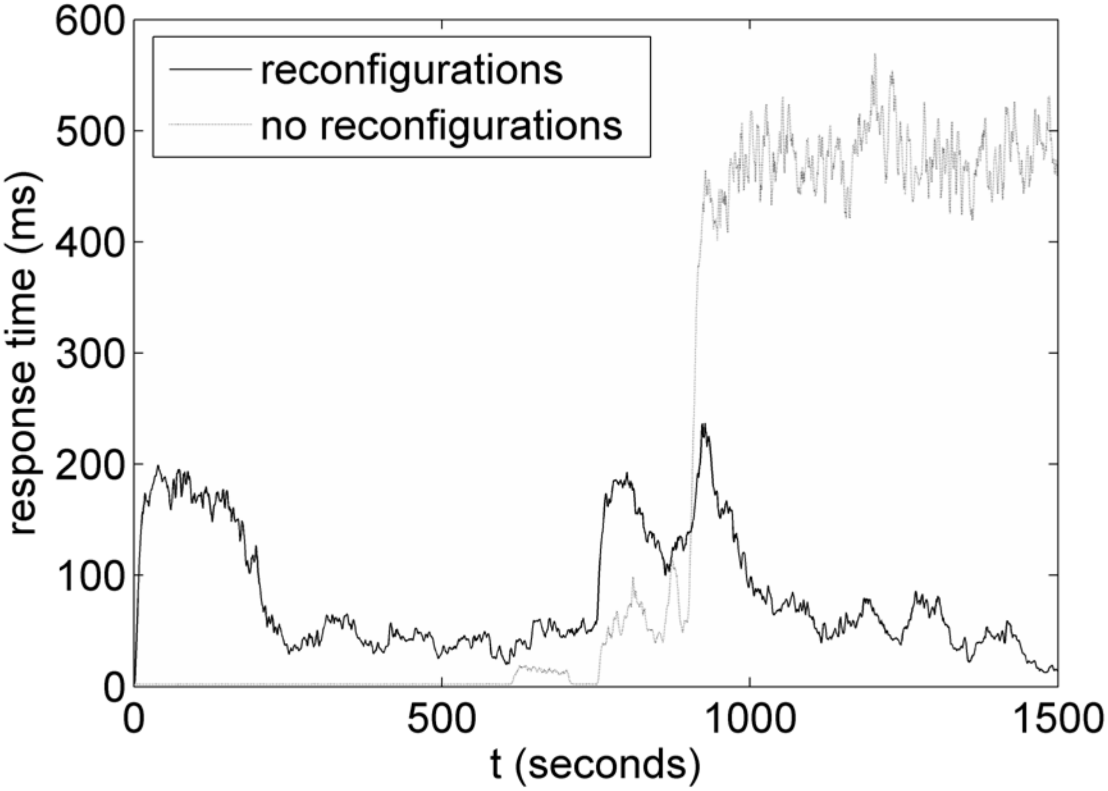


Figure 18. Overall mean response times for the simple experiment.

One conclusion and one conjecture are drawn from this relatively simple experiment. First the conclusion, tenant services must be rearranged as loads on those services vary significantly over time. This is demonstrated by Figure 18 where t>900. Second, the conjecture, if a configuration is found for a particular set of loads, e.g. t<600 that performs well, caching it for later use will decrease the amount of time it takes to find a configuration (no search algorithm is needed as it can be selected from the cache) and therefore should reduce response time. This is demonstrated by the left side, t<600, of Figure 18. The consistently higher values show that well performing configurations are not always found and switching to a new (and unproven) configuration every minute does not work as well as using one that has been proven to perform well.

Tenant Placement and Caching

Caching is a common tactic in on-line algorithms. Caching typically provides time savings at the expense of storage. The classic example is memory page caching of operating systems. Caching works well in this environment due to the locality of reference of memory requests by the applications hosted in the operating system. Although it works well over the average, its upper-bound suffers because a sequence of requests which lie outside the locality of reference will miss the cache every time.

The multiple resource tenant placement problem has nodes providing multiple resources and services consuming multiple resources. The multiple resource tenant placement problem is a multi-dimensional, multiple constraint knapsack problem. The problem can also be modeled as a vector packing problem. Both are NP-Hard. As described in [7], the problem is the immediate placement of static tenants, profiled *a priori* as they arrive. In the problem presented by Zhang et al services are not moved once placed. However, we generalize the problem in our work such that services not only can arrive but they can change size once placed, and furthermore services can be moved in response to changes. The online nature of this problem is not only the arrival but also the change in service size as time progresses. The Online Tenant Placement Problem in a real-time environment has time constraints. These time constraints are assumed to be on the order of seconds. Finding solutions to NP-Hard problems with long running binary integer programming or linear programming approximations is not acceptable. By the time a solution has been found, the changes to the service profiles render the profiles used in calculations obsolete. A fast heuristic must be employed.

The Cloud Chamber has the capability of employing various methods to find assignments of services to nodes. As described in Chapter 4, a greedy random heuristic in conjunction with a quality measure function determines the next configuration. The period between configuration changes is a tunable parameter. With the configuration of twenty five nodes described in the next section, the time to find the next configuration and communicate it to all nodes is accomplished reliably within 45 seconds.

Caching previously used configurations initially seemed a tactic worth pursuing, however, initial empirical tests and MATLAB simulations demonstrated otherwise. In the simulations, *every* previous configuration was made available in an unlimited size cache. Only occasionally was an entry in the cache of better quality than the quality of a configuration found using the heuristic. In the empirical investigations the choice to use an entry in the cache still required communication among the nodes and had less quality than a configuration from the heuristic.

The conclusion was to use a mixture of recently used configurations and random configurations as starting points, seeds, to the heuristic. The following theoretical and empirical analysis supports this conclusion.

Theoretical Analysis

The following definitions and theorems provide the theoretical basis for the usage of recent and random configurations as seeds to the random, greedy local search heuristic. Broadly, using the current configuration as a starting point for the local search minimizes the cost of change and often finds the best quality. Using random configurations for seeds finds better configurations. As the services are changing fast enough the current configuration is no longer any more useful than the random seed.

The environment consists of a matrix *N*, the set of nodes; a matrix *S*, the set of services; and an adjacency matrix *C*, a configuration mapping services in *S* to nodes in *N*. A service can only be mapped to a single node. Let $\mathbb{C}$ be the set of possible configurations. Specially, $\mathbb{C}$ is the collection of functions mapping *S* into *N*,

$$\mathbb{C} = \mathcal{F}\left(S, N\right)$$

Definition 9. (Hamming Distance) Given two configurations *C* and C^* in $\mathbb{C}$, define the Hamming distance $d(C,C^*)$ to be the count of drops (1 to a 0) and adds (0 to a 1) to change configuration *C* into configuration C^*.

As an example, if a service is moved, it is removed from a node and added to another, then the configuration *C* is changed to become configuration C^*; thus, the Hamming distance is two. The Hamming distance is a measure of dissimilarity between the two configurations.

Definition 10. (Quality functional) A quality function $q : \mathbb{C} \to \mathbb{R}^+$, is a functional that quantifies the quality of the configuration *C* for *N* and *S*. The lower the value $q(C)$ is, the better; and 0 is perfect quality.

The function maps each configuration to the $\mathbb{R}^+$ such that lower values are better and 0 is perfect. The Cloud Chamber currently uses the Provisioning Norm. It is an asymmetric norm with a tunable parameter allowing bias availability or performance. The Cloud Chamber for this chapter used α=0.99999 preferring performance.

$$\begin{aligned} q(N,C,S) &= \|N - CS\|_{\alpha,F} \\ &= (1-\alpha)\|(N-CS)_{+}\|_{F} + \alpha\|(N-CS)_{-}\|_{F} \end{aligned}$$

A random, greedy heuristic (defined formally in Chapter 4), *h*, uses the Provisioning Norm *q*, the nodes' resources, the services' resources, and an initial configuration. Given a configuration *C* to start the search, the heuristic returns a new configuration C^* such that the quality of C^* is the best found.

$$\begin{aligned} &C^* = h(q,N,C,S) \\ &q(N,C^*,S) < q(N,C,S) \end{aligned}$$

As described in Chapter 4, each iteration randomly selects a node for consideration. Each round of the heuristic has two phases. The first phase considers whether any service should be removed from the selected node. The service whose individual exclusion from the quality calculation improves the node's quality the most is removed. The second consideration is if any unassigned services would improve the node's quality. Each available service is individually included in the quality calculation. The one, if any, that improves the node's quality the most is assigned to the node.

A random configuration is noted as C_r. Applying the heuristic to C_r, *N*, and *S* yields C_r^* and is noted by

$$C_r^* \leftarrow h(q,N,C_r,S).$$

Similarly, the current configuration is noted as C_c. Applying the heuristic to C_c and a set of nodes and services is noted by

$C_c^* \leftarrow h\left(q, N, C_c, S\right)$

Figure 19 notionally shows the distances considered below. Distance (a) is the distance between the current configuration and a random configuration. Distance (d) is the distance between a random configuration and the configuration resulting from the heuristic. Distance (e) is the distance from the current configuration to its heuristic result. Distance (b) is the distance between the two resultant configurations. Finally, distance (c) is the distance from the current configuration to the configuration resulting from the heuristic starting at the random configuration.

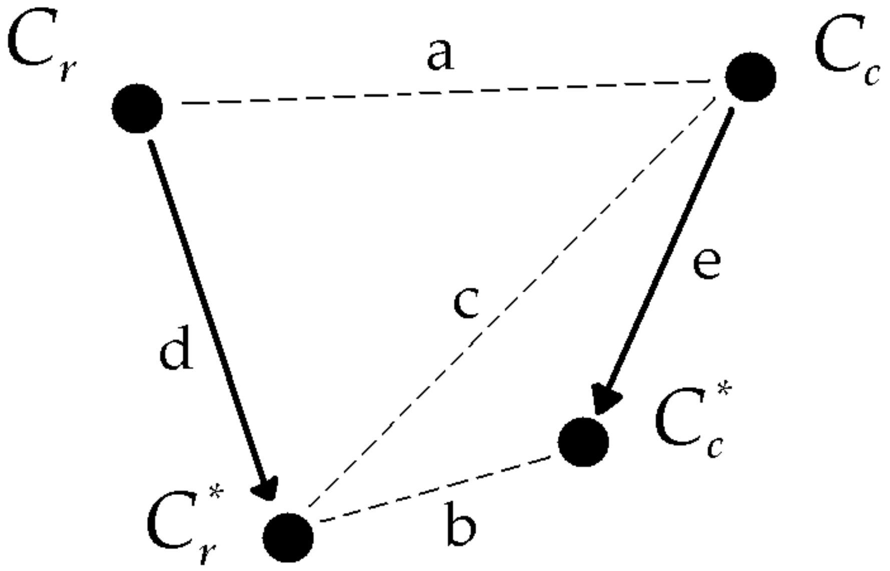


Figure 19. A two-dimensional representation of configuration space applying the heuristic to the current configuration and a random configuration. The distances a, b, and c are described in the theorems

Definition 11. Let $C, C^* \in \mathbb{C}$, let n be the number of nodes, and let $P\left[C_{ij} = C_{ij}^* = 1\right] = \frac{1}{n}$ denote the probability the *i*^th^ service is mapped to the *j*^th^ node in both C and C^*.

Definition 12. Let δ denote the change in the size of the services' resource profiles over time. The change from S to S_c is δ.

$$S_c = S + \delta$$

Assumption 1. Services are much smaller than nodes.

Theorem 8. Let s be the number of services and let $C, C^* \in \mathbb{C}$. The maximum value of the Hamming distance is

$$\forall C, C^*, \sup\left\{d\left(C, C^*\right)\right\} = 2s$$

Proof. If all services moved from their assigned node to a different node, each will be removed once and added once, for a total of two for each service.

Theorem 9. Let s be the number of services, n be the number of nodes, and $C, C^* \in \mathbb{C}$. The expected value of the distance ((a) in Figure 19) between two arbitrary distinct configurations C and C^* is

$$E\left[d\left(C, C^*\right)\right] = 2s\left(\frac{n-1}{n}\right)$$

Proof. If there are no services similarly assigned to the same node in C and C^*, then per Theorem 8,

$$d\left(C, C^*\right) = \sup\left\{d\left(C, C^*\right)\right\} = 2s$$

The probability of a service assigned to the same node in the arbitrary configurations *C* and C^*, is from Definition 11,

$$P\left[C_{ij} = C_{ij}^* = 1\right] = \frac{1}{n}.$$

Given this uniform distribution, the number of services expected to be assigned to the same node in both configurations is

$$E\left[\text{number of services assigned to same node out of } s\ \right] = s \cdot P\left[C_{ij} = C_{ij}^* = 1\right] = \frac{s}{n}.$$

For each of these services two moves (a remove and a add) are not required and thus lessening the Hamming distance, yielding,

$$E\left[d\left(C, C^*\right)\right] = 2s - 2\frac{s}{n},$$

and simplifying to

$$E\left[d\left(C, C^*\right)\right] = 2s\left(\frac{n-1}{n}\right).$$

Theorem 10. Let *s* be the number of service, $C, C^* \in \mathbb{C}$, and a single round is the dropping of a service and adding of a service. The expected number of rounds the *non-random*, greedy heuristic requires to settle is less than *s*.

Proof. The maximum distance between two arbitrary configurations, *C* and C^*, is $\max\left(d\left(C, C^*\right)\right) = 2s$. Transforming *C* into C^* requires dropping each service assigned in *C* and then adding each service as prescribed by C^*. If a round is defined as the dropping a service *and* adding a service, then $\leq s$ rounds are required to transform *C* into C^*. A service is dropped if one is available to drop. A service is added if one is available. The non-random greedy heuristic inspects all services to drop and drops the one that improves the quality the most. It similarly adds the one that improves the quality the most. In transforming C into C^*, all services are dropped until no more can be dropped and all services are added as they are available. Consider $d\left(C,C^*\right)=2s$; all services are assigned in C and none assigned as prescribed by C^*. In the first round, a service is dropped. In the bottom half of the round, the service is available to be added, and is. The process (dropping and adding of a single service) repeats for the rest of the rounds for a total of s rounds. A service must be dropped and added if available, by definition of the Hamming distance the number of rounds required to transform and C to C^* is the ceiling of half the distance: $\left\lceil \frac{d\left(C,C^*\right)}{2} \right\rceil$. And $\left\lceil \frac{d\left(C,C^*\right)}{2} \right\rceil \le s$.

If the heuristic randomly selects a node in each round, it may select a node with no services requiring a drop or an add. For the remaining theorems the amount of time the heuristic takes is not material. However for completeness, the expected number of rounds the random heuristic requires is derived in Theorem 11.

Theorem 11. Assume again, all services are assigned in both configurations. Let s be the number of services, n be the number of nodes, and $C,C^* \in \mathbb{C}$. The total number of expected node selections by the heuristic is

$$d\left(C,C^*\right)+\sum_{i=0}^{d\left(C,C^*\right)-1}\left\lceil \frac{1}{\left(1-\frac{d\left(C,C^*\right)-i}{2s}\right)^{\frac{s}{n}}} \right\rceil.$$

Proof. The probability of a service assigned to a particular node is $\frac{1}{n}$. The probability of a service needing dropped or added is $\frac{d\left(C,C^*\right)}{2s}$. When selecting a node at random, the expected number of services assigned to that node is $\frac{s}{n}$. The question needing answered is: what is the probability of selecting a node that has *no services needing moved*. Each round the selected node is expected to have $\frac{s}{n}$ services on it. The probability of a service *not* needing moved is $1-\frac{d\left(C,C^*\right)}{2s}$. The probability of selecting *exactly* $\frac{s}{n}$ services not needing moved is $\left(1-\frac{d\left(C,C^*\right)}{2s}\right)^{\frac{s}{n}}$. This is the probability of failure to select a node with *at least one* service. Thus the probability of success to select a node with at least one service is $1-\left(1-\frac{d\left(C,C^*\right)}{2s}\right)^{\frac{s}{n}}$.

Selecting the nodes using this probability is a Bernulli test yielding either a success or failure and thus a geometric distribution. The expected value of the number of consecutive node selections *without* at least one service to move is the inverse of the probability of failure,

$$\frac{1}{\left(1-\frac{d\left(C,C^*\right)}{2s}\right)^{\frac{s}{n}}}.$$

Once a success is had and implemented then the distance closes by one because the configuration is updated by one change from a 0 to a 1 or 1 to a 0. Let the updated configuration be C' and then $d\left(C',C^*\right)=d\left(C,C^*\right)-1$. Thus the *next* expected number of selections before a success uses C' and is

$$\frac{1}{\left(1-\frac{d\left(C',C^*\right)}{2s}\right)^{\frac{s}{n}}}.$$

A series of failures followed by a success is repeated until $C'=C^*$. The total number of expected failure selections is the summation of $d\left(C,C^*\right)$ expected values failed selections is

$$\sum_{i=0}^{d\left(C,C^*\right)-1}\left[\frac{1}{\left(1-\frac{d\left(C,C^*\right)-i}{2s}\right)^{\frac{s}{n}}}\right]$$

After each expected number of failures is a success. These successful selections are added in, $d\left(C,C^*\right)$, giving,

$$d\left(C,C^*\right)+\sum_{i=0}^{d\left(C,C^*\right)-1}\left[\frac{1}{\left(1-\frac{d\left(C,C^*\right)-i}{2s}\right)^{\frac{s}{n}}}\right].$$

Note from the earlier detailed description of the heuristic, the *depth* parameter is used to limit the node selections. In most of the experiments 200 to 500 were used. In general, most of the heuristic had settled before 100 selections.

Theorem 12. If the heuristic *h* is applied to an arbitrary C_r,

$$C_r^* \leftarrow h\left(q, N, C_r, S\right)$$
$$C_r^* \neq C_r$$

then the expected distance between C_r and C_r^* ((d) in Figure 19) is

$$E\left[d\left(C_r^*, C_r\right)\right] \leq 2s\left(\frac{n-1}{n}\right).$$

Proof. The distance between two arbitrarily selected configurations is $E\left[d\left(C, C^*\right)\right] = 2s\left(\frac{n-1}{n}\right)$. Here C_r is arbitrary but C_r^* is not arbitrary in relation to C_r, it is the product of the heuristic. Consider the services and their sizes. With some probability two or more services will be similar is size. Furthermore with some probability two services may add together to equal a third service, and so one. Therefore with some probability the heuristic will not move services. Thus, the expected distance between C_r and C_r^* must be less than arbitrary,

$$E\left[d\left(C_r^*, C_r\right)\right] \leq 2s\left(\frac{n-1}{n}\right).$$

Theorem 13. Using the current configuration as a seed to the heuristic yields a new configuration such that the distance from the current to the new ((e) in Figure 19) is a function of the change in the services' sizes, δ from Definition 12.

$$0 \leq d\left(C_c, C_c^*\right) = f\left(N, C_c, S^*, \delta\right) \leq 2s$$

Proof. If δ is sufficiently small, then the configuration will not change using the heuristic and the distance is 0; $C_c^*=C_c$ and $d(C_c,C_c^*)=0$. There exists a set, Δ_0, of changes to the set of services such that for all $\delta\in\Delta_0$, $S^*=S+\delta$ and $C_c=h(q,N,C_0,S)$ and $C_c^*=h(q,N,C_c,S^*)$ and $C_c^*=C_c$ and $d(C_c,C_c^*)=0$. Note: C_0 is whatever configuration seeded the heuristic to yield the current configuration C_c.

As δ increases in small amounts eventually a single service is dropped or added. There exists a set, Δ_1, of changes to the set of services such that for all $\delta\in\Delta_1$, $S^*=S+\delta$ and $C_c=h(q,N,C_0,S)$ and $C_c^*=h(q,N,C_c,S^*)$ and $C_c^*\neq C_c$ and $d(C_c,C_c^*)=1$. There exists Δ_m sets with $d(C_c,C_c^*)=m$ up to $m=2s$ where s is the number of services.

$$0\le d\left(C_c,C_c^*\right)=f\left(N,C_c,S^*,\delta\right)\le 2s$$

Theorem 14. The expected distance ((c) in Figure 19) of moving from the current configuration C_c to the result C_r^* of the random seed is

$$E\left[d(C_r^*,C_c)\right]=2s\left(\frac{n-1}{n}\right).$$

Proof. The expected distance between the current configuration C_c and the random configuration seed C_r from Theorem 9 (and (a) in Figure 19) is

$$E\left[d\left(C_c,C_r\right)\right]=2s\left(\frac{n-1}{n}\right).$$

The distance from the random starting point and its result of the heuristic search from Theorem 12 is

$$E\left[d\left(C_r^*,C_r\right)\right]\le 2s\left(\frac{n-1}{n}\right).$$

The Hamming distance triangular inequality provides

$$E\left[d\left(C_r^*, C_c\right)\right] \le E\left[d\left(C_c, C_r\right)\right] + E\left[d\left(C_r, C_r^*\right)\right].$$

Substituting and simplifying yields

$$E\left[d\left(C_r^*, C_c\right)\right] \le 2s\left(\frac{n-1}{n}\right) + 2s\left(\frac{n-1}{n}\right) = 4s\left(\frac{n-1}{n}\right).$$

The value can not be larger than $2s\left(\frac{n-1}{n}\right)$ per Theorem 8. The expected value will be no worse than arbitrarily selected configurations per Theorem 9. That is

$$E\left[d\left(C_r^*, C_c\right)\right] \le 2\left(\frac{n-1}{n}\right).$$

Experimentation

The Cloud Chamber is used for the empirical demonstration of seed caching with on-line heuristic searches in the tenant placement problem solution space. Specifically, the experiments contrast the performance of configurations discovered by searches using recent configurations as seeds versus those configurations discovered using random configurations as seeds. The setup of an experiment entails initially creating virtual nodes, deploying services, and generating a traffic load. The execution of the experiment is the execution of the traffic load. Response times, numbers of services moved, and the quality of configurations are collected.

Twenty five virtual nodes (running *lighttpd*, *serviceman*, and *nodeman* as described above in Chapter 3) are created in the Cloud Chamber. The capacities of ten of these 25 nodes are shown in Table 10. One hundred services are created from distributions as described above in section 3. A mix of distributions and parameters are used to create the 100 services. A sample of twenty-five services is shown in Table 11.

Traffic for the thirty minute trial was generated using a Poisson distribution [82]. To simulate changing traffic patterns experience by the set of services, this can not have a constant mean. If the mean of the Poisson distribution remains unchanged for thirty minutes, the profiles of the services remain unchanged. If the service profiles remain unchanged, reconfigurations do not happen as often because the system settles on one of good quality.

The traffic must provide a load on each service that drifts up, down, or both over the 30 minute trial. The overall load mean is around 10 hits per second per service. The mean of all traffic over time is plotted in Figure 20. A sample of a service's traffic overlays the mean in Figure 20. The Cloud Chamber handles over 1000 hits per second of traffic in this experiment. Heavier traffic loads were experimented with and can be handled. However at 2000 hits per second per service, limiting factors, such as single threaded web servers, result in a sufficient number of timeouts and http 500 error codes that skews results.

In order to create change and continue to use the properties of a Poisson process, the distribution was applied recursively to create self-similar traffic[83][84][85]. The thirty minute trial is divided into sixty thirty second segments. Each service has the hits per second, λ, assigned to each service segment. This provides an average hits per second for each service segment. Inside of each service segment, the assigned mean λ is the input to a Poisson distribution. From this second process each second in the segment is assigned a hits per second arrival rate.

As the experiment is executed, the traffic is fed to the *loadrunners*. The *loadrunner*s make the prescribed requests per second per service. While requests are being processed, performance data is collected. All nodes and *loadrunners* are rebooted. The process of executing the traffic and rebooting is repeated 30 times for each case.

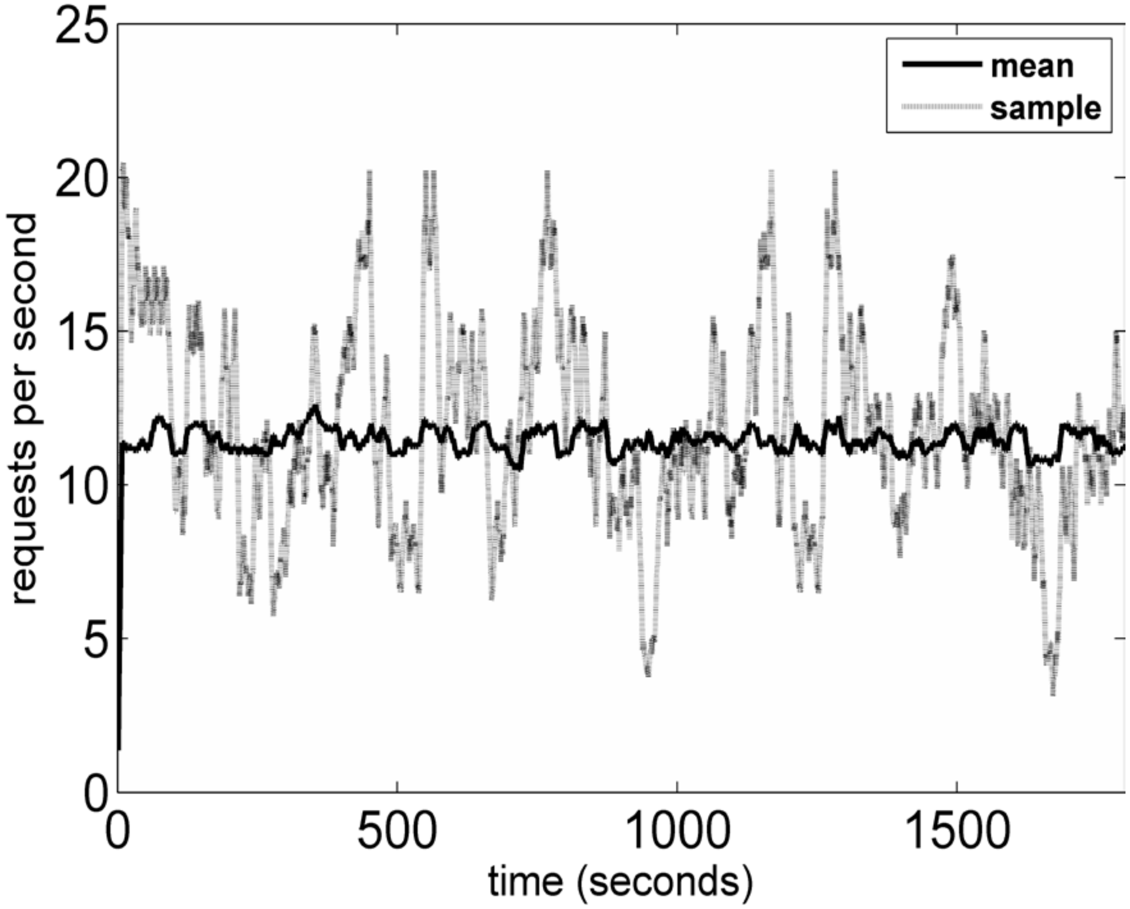


Figure 20. Mean traffic across all services over thirty minutes. An individual service is also plotted as a sample.

The data collected includes each node's operating system, web server, serviceman, and *nodeman*. Additionally, the *loadrunner* performance is captured. During the experiment nearly 10 million records were collected. In this experiment, three performance statistics are focused on: service response time, quality of the implemented configurations, and cost of change from the current configuration to the next configuration, e.g. cost of moving services.

The described testing framework is applied to seven test cases. The cases examine different starting conditions of the random, greedy, local search heuristic. The cases are described in Table 12. The starting conditions (configurations) are referred to as seeds. The first case examines starting the heuristic with five randomly generated seeds. The second considers using a single randomly generated seed. The sixth and seventh cases use the most recently used (current) configuration and the five most recent configurations, respectively. The middle three cases use a mix of random and recent. Case three has four random configurations and the current configuration. Case four uses a single random seed and the current configuration. Case five has a single random seed and four most recent configurations. In another, four random seeds plus the most recent configuration are used. And finally, five random seeds are considered without any current or recent configurations.

Table 12. Experimental Cases: Cache Seed and Random Seed Combinations.

Case Number	Cache Seeds	Random Seeds
1	0	5
2	0	1
3	1	4
4	1	1
5	4	1
6	1	0
7	5	0

To summarize, seven test cases will be run in succession. Each case changes the type and number of seeds used by the heuristic. Each case is executed against the same traffic load 30 times.

Results

The results are discussed here in four sections. Each discussion considers the results from the perspective of grouping the cases into three categories: random, cache, and mixed. In Table 13, the cost of moving services during reconfiguration is presented in the first set of columns, followed by the overall quality of the configurations in the second set columns, and the performance as response time in the third set of columns. Finally a few interesting miscellaneous findings are presented.

Table 13. Cost, Quality, and Response for All Random & Cached Seed Cases.

(μ = mean and σ^2 = standard deviation)

Case	Cost of Change		Quality (x10^8)		Response Time	
	μ	σ^2	μ	σ^2	μ	σ^2
1	191.3	3.9	47970.5	193.6	159.48	344.19
2	191.6	3.8	48069.8	123.6	135.46	298.86
3	98.8	93.9	47901.4	202.0	136.55	341.19
4	56.1	84.3	47881.3	234.1	87.27	369.98
5	102.1	93.5	47920.1	186.4	129.53	333.07
6	3.0	1.7	47961.6	122.5	121.03	332.96
7	5.8	6.6	47498.0	310.0	155.92	422.92

Cost of Change

Cost of reconfiguring derives from the movement of services. In this experiment consideration is only given to a service that is dropped from a node or added to a node. Future iterations of the Cloud Chamber could give consideration to the cost of moving any state or files associated with the physical move through a simple modification to the optimization cost function. At the end of epoch (described in Section 5), records about the new configuration were logged. These records included the quality of the newly selected configuration and the Hamming distance between the previous configuration and the new configuration. The configuration is represented by an *n* x *s* adjacency matrix. The Hamming distance is calculated as the sum of the absolute values of the difference between the two matrices. In MATLAB syntax,

$$sum\left(sum\left(abs\left(C_n - C_{n+1}\right)\right)\right).$$

Table 13 shows overall mean cost of change for each case across thirty trials. Figure 21 shows cumulative mean cost of change over time for the seven cases. The cost of change is the lowest in cases using only cached seeds and is the greatest in cases using only random seeds.

Theorem 9 states the expected distance from a random seed to its heuristic result to be

$$E\left[d\left(C, C^*\right)\right] = 2s\left(\frac{n-1}{n}\right).$$

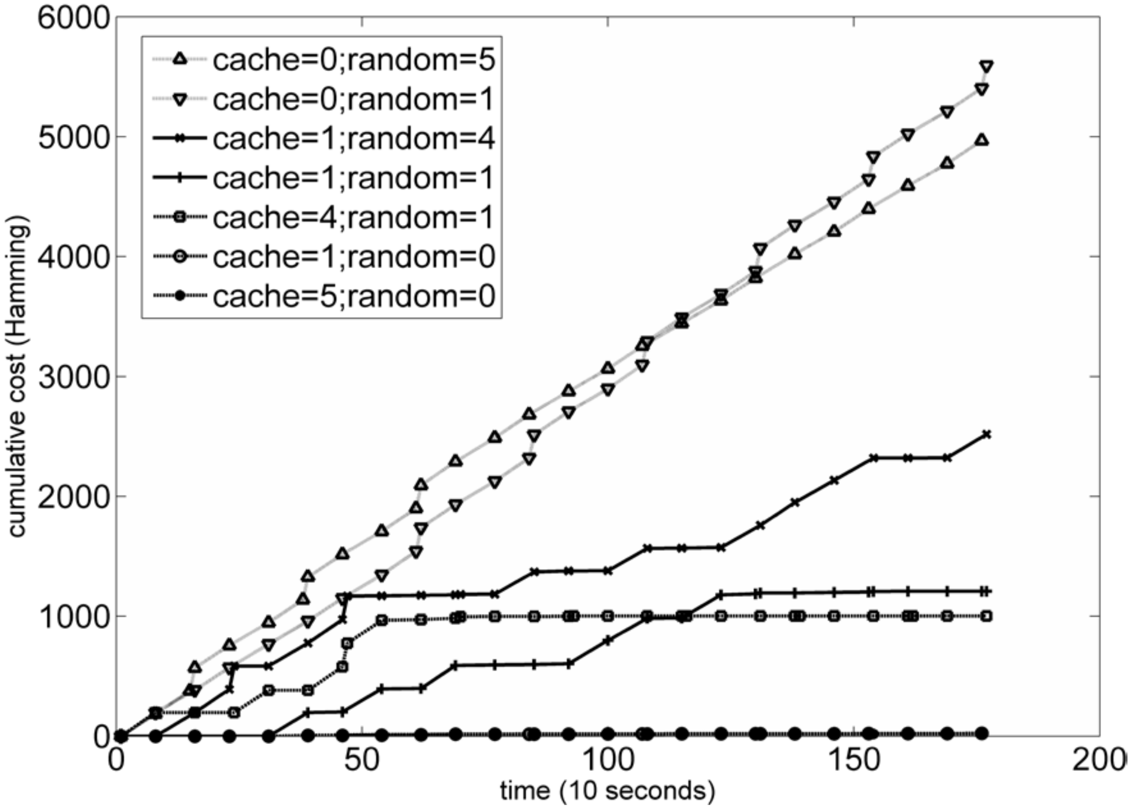


Figure 21. Cumulative Cost of Change.

For random seed cases 1 and 2, the expected value would be 192. Figure 21 and Table 13 both reflect this. Clearly including cached seeds reduces the cost of change. Additionally including no random seeds reduces the cost of change substantially (to near zero).

Consider Theorem 9, if the change experienced by the services is small, the new configuration should come from the cache seed instead of the random seed. Recall from Assumption 1 and Definition 12, δ is the change in the service profile from one time step to the next. The difference, δ, was approximated as the sum of the absolute values of the difference in the service resource values from one time step to the next.

Table 14 empirically demonstrates Theorem 13. The three mixed cases with both cached and random seeds are presented. Each case, across all thirty trials, had a total number of reconfigurations. This value is presented in the column labeled “ALL” with the total number of small changes (δ<0.12), total number of large changes (δ>0.24), the overall mean of δ, and its standard deviation. The reconfigurations were a result of executing the heuristic with either a random seed or a cached seed. The breakdown of each is presented in its respective column.

Table 14. Small & Large δ Seeds: Cache vs. Random in Mixed Cases

Case		**all**	**cache**	**random**
3 cache: 1 random: 4	count	895	650	245
	δ <0.12	218	205	13
	δ >0.24	170	94	76
	μ	0.1800	0.1647	0.2204
	σ^2	0.084	0.078	0.0885
4 cache: 1 random: 1	count	943	853	90
	δ <0.12	45	45	0
	δ >0.24	201	170	31
	μ	0.1889	0.1837	0.2382
	σ^2	0.085	0.0801	0.112
5 cache: 4 random: 1	count	919	817	102
	δ <0.12	232	227	5
	δ >0.24	183	146	37
	μ	0.1814	0.1754	0.2293
	σ^2	0.088	0.086	0.089

The first point of interest is in all three cases the likelihood of the cache seed resulting in the new configuration is 3 to 10 times higher than the random seed. This reflects that the services are changing such that the new service profiles are dependent on the previous service profiles. This is true even in case 3 where the number of random seeds is 4 and cached is 1. Referencing back to Figure 19, the distance (e) is generally smaller than the distance (c).

Second, for the purposes of this illustration, values of $\delta<0.12$ are considered low. When $\delta<0.12$, the likelihood of a new configuration resulting from the random seed reduces in cases 3, 4, and 5. When $\delta>0.24$, the likelihood of a new configuration resulting from the random seed increases in cases 3, 4, and 5.

Third, note the mean δ across each case for all of trials: 0.1800, 0.1889, and 0.1814. In all three cases, the mean δ for random seeds is higher than the overall mean while the cache mean δ is lower than the overall mean. This behavior implies that small changes show preference for the use of recent configuration seeds. Furthermore, when large changes are expected employing random seeds is beneficial.

Quality

The quality of a configuration as defined in Definition 10 is a single real value derived from a configuration for a specific set of nodes and services. This single value represents the quality of the entire system. If services are assigned to nodes such that no more and no less than 100% of the nodes' resources are consumed, the quality function is zero, e.g. perfect placement. If the service demand is zero, then the quality function will be maximized. Conceptually, one can consider the quality as the Euclidean distance to the perfect configuration. In searching for the next configuration, the heuristic looks for the best quality configuration (the lowest value).

As shown in Table 13, the mean quality from case to case does not vary significantly. However cases 3, 4, and 5 utilize both cached and random seeds and were lower than the non-mixed cases 1, 2, 6. Case seven (5 cached recently used configurations), came in with the over all best. In the mixed cases 3, 4, and 5, if δ was small enough, a cached seed would most likely provide the best solution. If the δ were large enough, the random seed would provide additional search possibilities. In the non-mixed cases 1, 2, and 6, regardless of the size of δ, the seed came from the cache for 6 and 7 and random for 1 and 2. Further investigation into why case 7 came in so much lower than all of the other cases is required.

Figures 21, 22, and 23 show quality over time. The x-axis is 10 second windows. The y-axis is the mean quality across all thirty trials for the specific 10 second window. Recall the same traffic load was applied in each trial. Initially the service profiles are near zeros and as they are modeled they grow, thus reducing the quality as time progresses and better configurations are found.

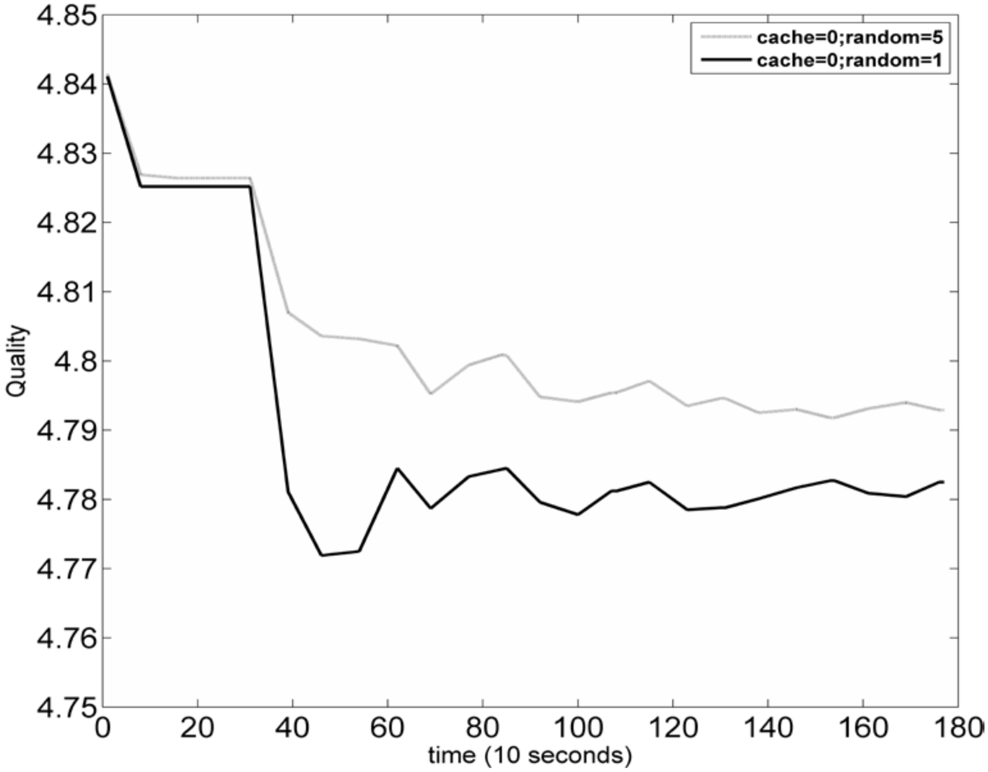


Figure 22. Configuration Quality with random (Cases 1, 2).

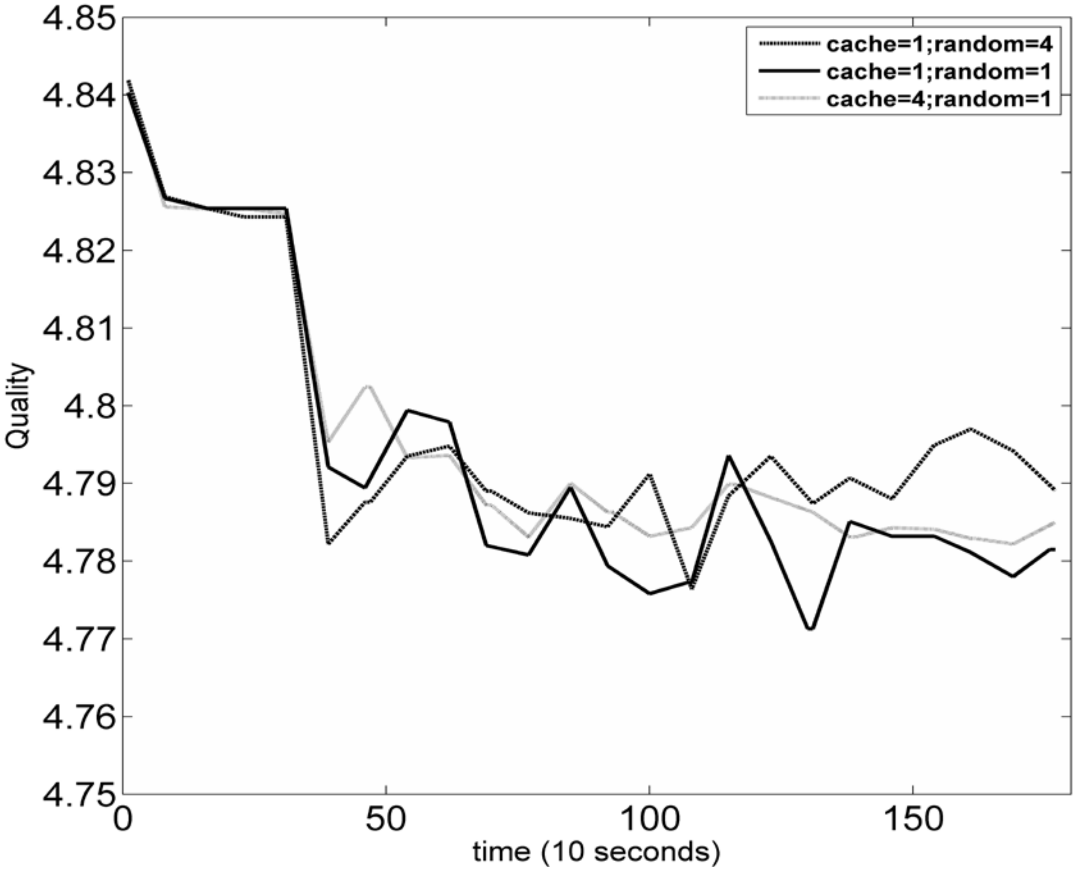


Figure 23. Configuration Quality with mixed (Cases 3, 4, 5).

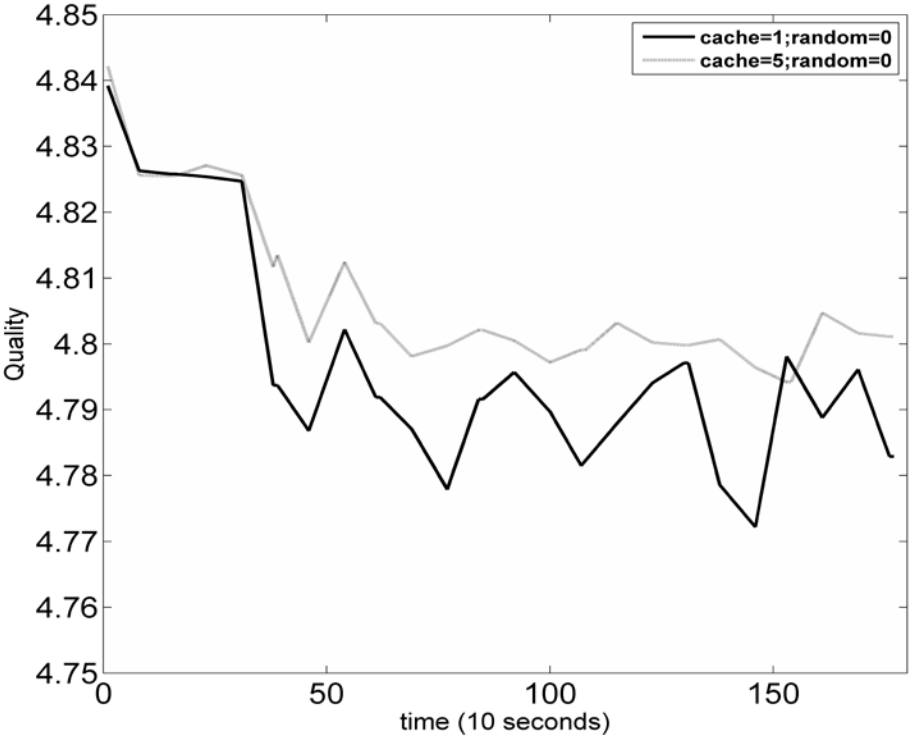


Figure 24. Configuration Quality with caching only (Cases 6, 7).

Response Time

Response time for this experiment is the mean of the response time from the moment the request is issued to the moment all of the data is returned successfully (http code 200). If the http connection failed to complete before 10 seconds elapsed, the connection was terminated and 10 seconds was used for that connection. Any unsuccessful http request (http codes not equal to 200) were not considered in the response time. Figures 24, 25, and 26 show the response times over time for all the cases.

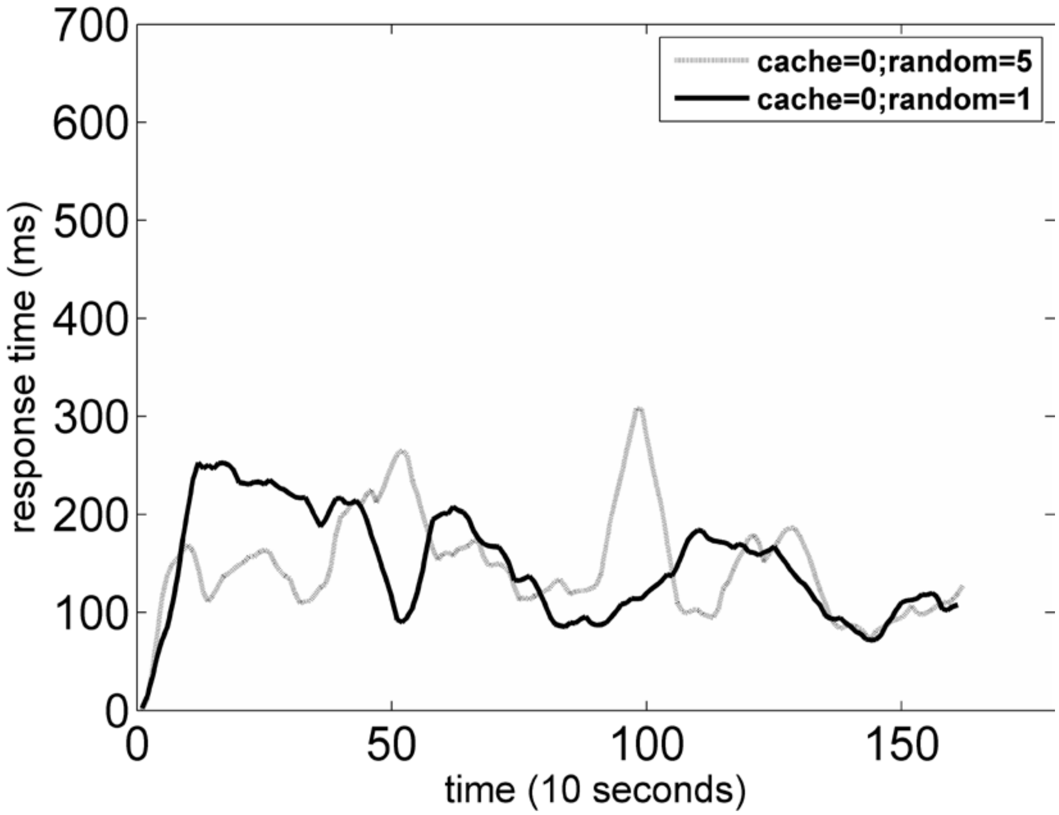


Figure 25. Mean response time for random. (Cases 1, 2).

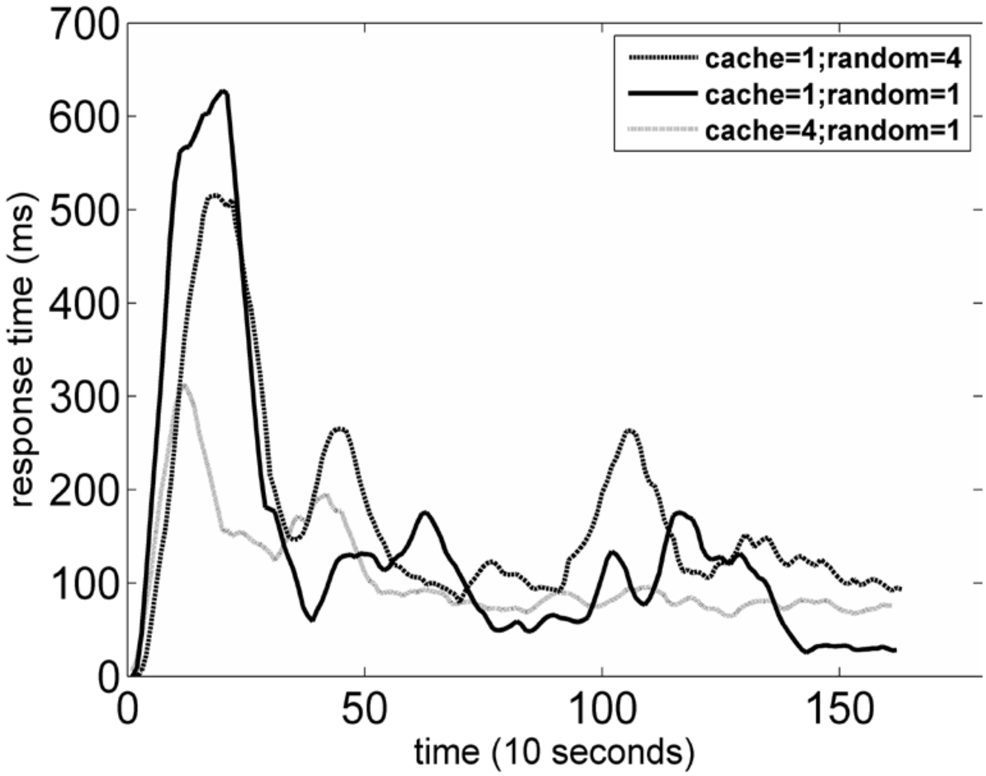


Figure 26. Mean response time for mixed (Cases 3, 4, 5).

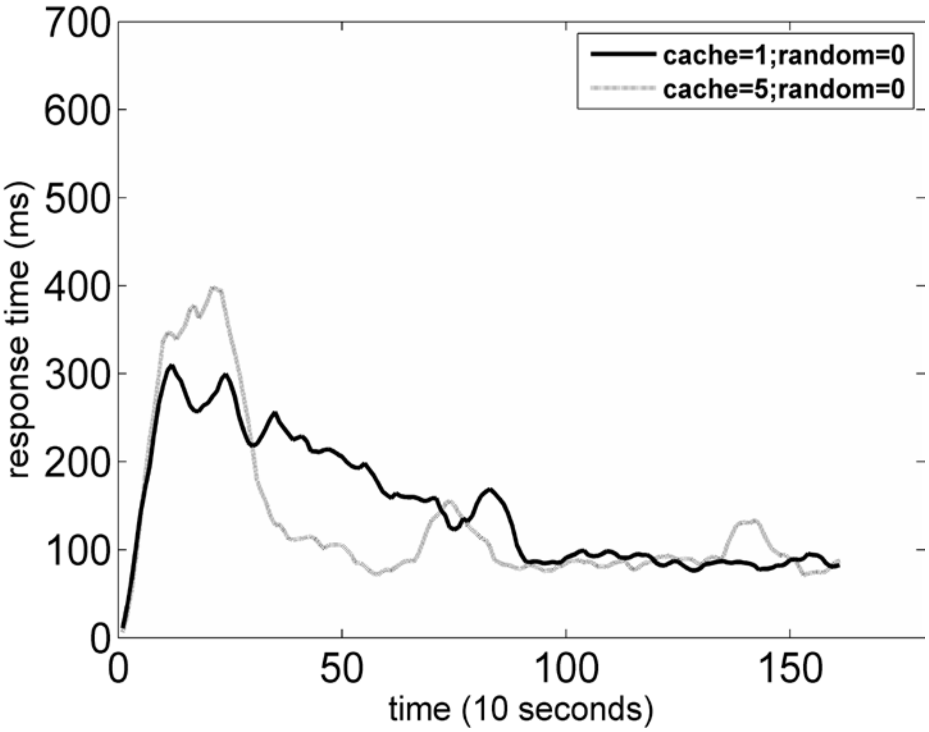


Figure 27. Mean response time for caching only (Cases 6, 7).

Table 13 shows case 4 has the best response time by a significant margin. Furthermore, the cost of change is the best of any with random seeds. Case 4 has a single cached configuration: the current configuration. If the random seed results in a better configuration, this better configuration becomes the current configuration via the heuristic, and the previously cached configuration is then lost. In contrast, case 5 has the five most recently used configurations. Using recent configurations as seeds *and* a random seed can create thrashing. The random seed yields the best configuration, a new (previously unseen) configuration. Then a previously used configuration seed yields the best (one similar to the seed). This jumping between two configurations that are not very similar results in services being moved over and over.. Case 4 does not have this because the most recently used configuration is replaced every step.

Summary

The Cloud Chamber provides a unique testbed to witness the behavior of tenant web services over time under varying loads. The results in this work indicate to minimize the movement of services, the search for the next configuration should only begin from the current or recently current configurations, case 6 and 7. The results further indicate if the performance is more important than service movement, the search should start at the current configuration and a randomly selected position, case 4.

In conclusion the Cloud Chamber meets all the requirements described in the introduction. The Cloud Chamber creates a facility to create, exercise, and examine the behavior of tenants in a Software as a Service environment. Services of various shapes and sizes can be deployed onto a heterogeneous set of nodes providing different amounts of resources. These services can be executed with any size of prescribed load for any length of time. The nodes self-organize autonomously finding and implementing tenant assignments in response to changes in the environment. The author is unaware of any such facility.

The Cloud Chamber provides an environment for rich research in the area of the tenant placement problem and general tenant behavior. This work presents a generalized form of the on-line tenant placement problem where services change size and can be moved once placed. This work analytically and empirically demonstrates cached and randomized initial conditions for the heuristic search.

VI. Performance Feedback Loops in Self-Organizing Web Servers

Introduction

System administrators require controls and configuration settings for implementing management policies on their systems. The scale of these systems has grown to thousands of servers. Manually changing settings on individual systems at this scale is unfeasible. Considering changes to thousands of individual systems is challenging and can require too much time to be effective. Autonomic processes to self-monitor and self-configure systems are a requirement for system administrators today. This work considers self-monitoring and self-configuring as a solution to these challenges.

Arranging a set of web services on a set of web servers under multiple resource constraints is known as the on-line tenant placement problem [7]. This placement of services on servers is, in the most general sense, a multi-dimensional knapsack problem, otherwise known as a vector packing problem. This work specifically generalizes the vector packing problem by allowing services to change size and optionally to move after they are placed. This generalization is critical because the demand (requests per second) placed on a service changes over time and thus changes the resources the service demands of the server. If services are unable to move as their demand increases, performance suffers.

Service resource profiles are modeled online using measurements of processor, memory, bandwidth consumption, and requests per second. The model yields service resource profiles, a reservation of computing resources. These profiles do not predict performance, and to address this problem this work enhances the service movement by incorporating a feedback control based on the services’ performance. When the size of the service resource demand is strictly a function of its requests per second, the service may fit on a node mathematically. However, the performance of the service may still suffer. This work proposes a proportional integral derivative (PID) controller using the performance error to calculate a factor by which the individual service’s profile is multiplied. This profile coefficient grows as the performance error grows. The performance error is the response time minus the specified threshold.

Performance of web services is subject to many factors such as but not limited to other services on the server, operating system overhead, request queue management, and network delays. For example, for each handled request the service’s resource usage can be measured; however the operating system overhead specifically associated with the individual request can not be accurately measured. This overhead includes tasks such as memory page management, Transport Control Protocol send buffer management, and CPU context switching. Furthermore, the server likely houses other web services and other tasks unrelated to the web services under investigation.

These errors, or overhead, in modeling the service resource profile in relation to performance are reflected in the profile coefficient. As the performance of the service suffers, the service resource requirements are increased. This adjustment of the service resource profile allows for more accurate coordination of profiling and performance in three ways. First, the service profile may reach a sufficient level to cause the profiled service or another service on the same node to be moved to another node, thus increasing the resources available on that node. Second, augmenting the profile by the inclusion of overhead external to the service itself should improve performance. Third, the effective profile may more accurately reflect the true resource demands of the service beyond those capable of being captured with the current sensors. Presented in this work are the theoretical basis and empirical results for implementing the performance feedback control. The empirical results show controlling service profiles improves performance. For the particular experiment presented here, the controller improves performance up to 66%, reduces queue sizes by 66%, and continues to include all services. Works related to the control of web servers and vector packing are discussed in the next section. The theoretical basis for this work follows the related work. Finally, the empirical testbed setup and results are presented.

Related Works

The application of formal control theories to the performance of web servers dates back ten years ago. The general model is to define a performance target such as number of requests per second (throughput), client response time (performance), or the amount of time the request remains in the queue. As the actual performance varies from the target performance, the system adjusts an internal setting of the web server in an attempt to bring the actual performance closer to the target performance. Primarily, two control mechanisms were used: admission control and/or processing allocation. As performance varies away from the target, more or less of the control is applied. This is generally referred to as a proportional feedback control loop.

The earliest formal works found addressing control theory to web servers is Abdelzaher, et al.[86] and Lu, et al.[87]. The first work generally describes a web server as a linearly time varying system with sensors and actuators. They model and empirically test a feedback control loop demonstrating the ability to guarantee times. The second uses a feedback loop with proven performance guarantees based on established analytical methods, specifically Root-Locus method.

These conference proceedings led to a more robust work of Abdelzaher, et al.[88]. To the proceedings they add service differentiation and overload protection. They divide the requests into two classes: premium and basic. The premium services' guarantees are supported at the cost to the basic class demands. The actuators discussed included admission control, collectively sharing degraded response times, and content control. The later includes fewer objects in a page by excluding pictures and videos. Content control also removes links to further internal content lessening the likelihood of users loading another page.

Their environment is a physical web server hosting multiple virtual web sites. Each site offers different guarantees to multiple classes of clients. Their model uses two actuators: one to limit the included content and a second to deny client admission. Their results using an altered Apache web server show performance can be controlled by limiting included content and client admission.

A second early body of work culminated in the book [89] by Hellerstein et al. This work is based on [90][91][92][93][94]. The work discusses feedback control of computing systems covering modeling, proportion control, PID, and state space. Throughout the work, two specific examples are referenced. An apache web server's performance is controlled by adjusting the maximum number of concurrent users and the session timeout value, both independently and together. The second example is controlling the number of remote procedure calls by adjusting the maximum number of users. For each example formal control models are created to illustrate the various aspects on control theory. In the earlier works, performance is improved using these controls.

In more recent works, Heo et al [95] save energy controlling the number of servers in the overall system, the voltage on individual processors, and backup requests. They use these controls in centralized, decentralized, and Multiple Input Multiple Output (MIMO) feedback control models and compare the results. Kjaer et al [96] show that off–line estimations can be problematic in online applications, and they propose a Predictive Feedback Controller with parameters estimated online. Their results are derived from an Apache testbed server and simulations. They allocate additional CPU time to control the overall response time. Xu et al [97] similarly use allocated processor time to different classes of web pages to control the end-to-end response time using their eQoS framework. Urgaonkar et al [27] show over-booking CPU reservations increases a Linux cluster utilization substantially; 5% overbooking yielded 300% increase in utilization.

Epstein et al [14][98][99] first present a general framework for vector assignment problems assigning *n* input vectors to *m* machines given a target function to minimize. The general approach presented enables a polynomial time approximation scheme (PTAS) for a wide class of target functions. The second work [98] looks at the oriented multi-dimensional dynamic bin packing problem for two dimensions, three dimensions and multiple dimensions. Specifically, the dynamic packing of squares and rectangles into unit squares and dynamic packing of three-dimensional cubes and boxes into unit cubes are presented. Her third work considers the bin packing problem such that the items can be moved. Selfish agents own items. Each agent is charged with a cost according to the fraction of the used bin space its item requires. The cost of the bin is split across agents proportionally. The selfish agents prefer their items packed in a fuller bin. The overall goal however is to minimize the number of the used bins.

Methodology

The problem defined in this work is an online vector packing problem. The vector packing problem is *n* multi-dimensional vectors to be packed into *m* multi-dimensional vectors, i.e. items into bins. Various constraints can be added such as minimizing the number of bins or equalizing the load of each bin. Some of the problems consider the vectors in a geometric sense as the placed vector consumes two or three dimensions of space, e.g. the dimensions are dependent on one another. Others allow the dimensions to be independent of each other. The novelty added in this work to the previously well defined vector packing problem is (1) once the items are placed the items can move and (2) the item vectors and bin vectors change size over time.

Web services consume computing resources provided by the web servers (nodes) on which they execute. A web service resource profile is a vector of values representing this amount of consumption. A node resource profile is a similar vector of values representing the amount of resources the node provides. Each entry in the vectors represents computer resources such as CPU, memory, and bandwidth. As services are placed on a node, the service resource profile vector is packed into the node resource vector. Over time service resource profile vectors change in response to the changing demands of the services. A utility function ranks the quality of the configuration of services on nodes. As services change in size, the quality of the configuration changes, and a new, better configuration is sought.

The above model does not consider the actual performance experienced by the consumers of the services. This experience is typically described using response time. Often demands of service quality are defined in a service level agreement (SLA). These service level agreements define a threshold, such as 100 ms, that invokes a fiscal penalty of some sort if exceeded. This work focuses on incorporating performance feedback in the form of a SLA. This threshold and the services' performance will be used to affect the above model to influence changes in the configuration.

Definition 13. An *actual* profile is a vector of measured and normalized resource values of a service or node.

For example, a node is defined (measured) to have a specified processing capacity, memory capacity, and bandwidth capacity. A service's resource consumption is measured as the service is processed.

Definition 14. An *effective* profile is a vector whose values are derived from the adjusted actual profile and which is used for all calculations and considerations.

In other words, the appearance of the node's resource supply and the services' resource demands to the system can be adjusted. The adjustment allows for policy considerations, e.g. under-performing nodes or services can have their profiles adjusted appropriately.

Definition 15. *Acceptable performance* is defined as a measured response time and its corresponding condition. For the rest of this work, acceptable performance is less than 100 milliseconds as measured by a service's mean response time over the last 10 seconds.

Definition 16. Define $\blacklozenge$ to be a product such that $M^* = \vec{v} \blacklozenge M$ where $\vec{v}$ is a vector of length *m*, M is a *m* by *n* matrix and M^* is a *m* by *n* matrix where $M^*_{ij} = \vec{v}_i \cdot M_{ij}$. In other words, the *i*th entry in $\vec{v}$ multiplies the each entry in the *i*th row of *M*. More formally it is defined as $\vec{v} \blacklozenge M \equiv \vec{v}^T \otimes \mathbf{1}^T \odot M$.

Definition 17. The actual, S, and effective, S^*, service matrices are matrices respectively comprised of all the actual and effective service resource vectors.

Definition 18. The service adjustment vector, $\vec{\Delta}_s$, is an array whose elements represent the per service multiplicative adjustment that translates the actual service profiles into the effective service profiles using Definition 16. The values are by default 1. Values larger than 1 make a service appear larger. And thus, the effective node matrix is $S^* = \vec{\Delta}_s \blacklozenge S$.

Definition 19. The actual, N, and effective, N^*, node matrices are matrices respectively comprised of all the actual and effective node resource vectors.

Definition 20. The node adjustment vector, $\vec{\Delta}_n$, is an array whose elements represent the per node multiplicative adjustment that translates the actual node profiles into the effective node profiles using Definition 16. The values are by default 1. Values smaller than 1 make the node appear smaller. And thus the effective node matrix is defined $N^* = \vec{\Delta}_n \blacklozenge N$.

Definition 21. Let C be the adjacency matrix mapping services to nodes.

Definition 22. Let $N^* - CS^*$ be the effective residual resources.

Definitions 7 and 8 are included for generality. The rest of this work does not consider controlling the effective node resource profiles and is left for future work. For Definition 22 in the rest of this work $N^* = N$. The effective node resource profile is the actual node resource profile.

The actual service profile matrix is built from measurements of each service's resource consumption. Each service request's raw consumption values for CPU, memory, and bandwidth are logged to a rotating buffer. This raw data, using linear regression, creates a linear model, as a function of hits per second, for each resource for each service, e.g. 3 resources and 100 services is 300 linear models. The value placed into the actual service profile is calculated from its model using hits per second as its input. Each entry is the following equation where *i* is the *i*th service and *j* is the *j*th resource.

$$S_{ij} = m_{ij} x_i + b_{ij}$$

The hits per second can be predicted in a variety of ways depending on how much forecasting data is available. For the experiments in this work the moving average of the last 20 seconds for service *i* is used for x_i.

The values in the actual service profile matrix can change over time. Conceptually one assumes the internal logic of a service does not change over time. However the inputs a service uses in its internal logic could change over time, e.g. a stock exchange service might experience batches of predominant sells and later batches of predominant buys. The internal logic remains the same however flowing differently, and thus, the measured and modeled resource consumption m_{ij}, b_{ij} may then change measurably over time.

Consideration must be given to changes to the values in the actual service profile matrix. In general for this work, the services do not vary substantially between each successive call and thus m_{ij}, b_{ij} do not change over time significantly. Furthermore, the overhead of the service resource consumption b_{ij} is not the dominant term, e.g. $m_{ij} > b_{ij} > 0$. This follows from the general assumption that these services are stateless; each call is independent of each other. Another assumption is the resource consumption of a single service request is much smaller than that of a node's actual resources. For example, a single call to a service will only consume a small percentage of the server's CPU. These assumptions lead to the fact the changes in the values in the actual service profile are linked to the changes in the number of hits per second. Steady traffic loads yield steady profiles.

The values in the service profile matrix represent a resource reservation. This representation is multi-dimensional, not simply CPU but also memory and bandwidth. The prominent research in resource scheduling control uses a one dimensional model. In the work presented here, the multi-dimensional reservations are derived from a linear model and, as discussed above, are driven by the load placed on the service in terms of hits per second. Once the actual service profile has been created from the linear model, the underlying mechanism of the model has been abstracted, e.g. the value in the profile. Other models could be considered. The method could be changed to a non-linear model or include other sensors, without loss of generality. For example, certain operating system overhead such as context switching, memory page management, and TCP control can be measured but are difficult to directly attribute to a service executing in an application engine such as a web server or managed environment like Java and .NET. Therefore in reality, additional CPU, memory, and bandwidth are used by the service but can not be attributed to the service. If it could be attributed, this inflating influence on the reservation could be inserted to the model.

Further utilizing the abstraction of the service profile, feedback from the performance of each service is applied to its profile. This transformed profile is the effective profile. As the performance of a service suffers, the reservation represented by the actual service profile is increased. For each service a threshold of acceptable performance is defined as a service level agreement. In this work the threshold is set at 100 milliseconds. For each service the response time is collected each second. Once the time for a reconfiguration is reached, each service is checked for performance compliance every minute. If the service is out of compliance, the feedback multiplier is adjusted. Specifically the vector $\vec{\Delta}_s$ contains the multiplier for each service. Each starts as 1.0. The effective service profile is $S^* = \vec{\Delta}_s \blacklozenge S$. Once the service moves to a new node, the value in $\vec{\Delta}_s$ is set to 1.0. As demonstrated in [87][88][90], control theory can be applied to formally derive this adjustment value on a single web server. The purpose of this work is to neither duplicate their work nor demonstrate this formality. The purpose of this work is to show that in a multi-server, decentralized, distributed, self-organizing system, integrating a mathematical representation of performance policy and mechanisms to implement the policy improve response time throughout the system. As described next, a Proportional Integral Derivative (PID) control model implements policy, although optimization of its various parameters is left for future work.

The PID controller takes the service response time as an input. Subtracting the service level agreement threshold (100ms) from the response time produces the error. If the error is less than zero, then the adjustment value (from here forward referred to as the *coefficient*) is 1.0. Otherwise, the error drives the PID. As defined below, the PID is the sum of three components: the proportion $K_p \cdot e_i(t)$, the integral $T_i \cdot \left[e_i(t)+e_i(t-1)\right]$, and the derivative $T_i \cdot \left[e_i(t)-e_i(t-1)\right]$, each weighted with a parameter. The proportion is how much the current error should influence the adjustment. The integral is how much the recent errors should influence the adjustment. The derivative is how much the rate of change in the error should influence the adjustment. The weight of influence of each component is subject to significant research, and optimality is dependent on the particular application. The Zeigler-Nichols method [100] was developed in the 1940's and is considered a sufficient method for tuning parameters. The weights are 0.6, 0.5, and 0.125 respectively. The following outlines the implementation for the i^{th} service with error $e_i(t)$ at time t.

$$K_p = 0.6$$
$$T_i = 0.5$$
$$T_d = 0.125$$

$$\vec{\Delta}_{s,i} = \frac{K_p \cdot e_i(t)+T_i \cdot \left[e_i(t)+e_i(t-1)\right]+T_i \cdot \left[e_i(t)-e_i(t-1)\right]}{threshold}$$

The threshold is the service level agreement value of 100ms. If the current response time is 500ms, the current error is $e_i(t)=400$. Assuming $e_i(t-1)=0$, then

$$\vec{\Delta}_{s,i} = \frac{0.6 \cdot 400 + 0.5 \cdot 400 + 0.125 \cdot 400}{100} = 4.9$$

would lead to an effective profile of nearly 5 times the actual profile for the i^{th} service. The controller affects the coefficient by translating each service's actual profile to its effective profile.

In using response time as a feedback control, the composition of response time must be considered. Response time of a service is the sum of network latency, queuing time, and service time. The response time is the time the request enters the system and the response leaves, and thus, latency is out of the scope of this work. In the testbed described below, the request and response travel through a virtual switch, a physical switch, and a second virtual switch and is thus negligible. In terms of queuing theory, the *actual* service profile is a multi-dimensional representation of the queue service time and is independent of the queuing time. The *effective* service profile indirectly reduces the queuing times across the larger system by reserving additional resources. This is demonstrated in the later results in the reduction of response times due to the system arranging the services across nodes to more efficiently use the resources or potentially exclude services that are too demanding. In [89], the control directly reduces the queue size and therefore manages performance by rejecting requests.. The control presented here indirectly reduces the requests in the queue.

Control of the effective profile is further extended to nodes. The control of nodes presented here is not considered in the theoretical basis or empirical results and is included for generality. If the nodes are housing services that are out of compliance, the profile of the node is reduced. Reducing the amount of resources of a node mathematically increases the likelihood a service is moved from that node. The effective profile is reduced in proportion to the magnitude of the housed services' errors. For the nodes, the error is the mean of all services' errors, including those that are zero. Here the j^{th} node has m services, and the node coefficient is calculated.

$$K_p = 0.6$$
$$T_i = 0.5$$
$$T_d = 0.125$$

$$e_j(t) = \frac{\sum_{i}^{m} e_i(t)}{m}$$

$$\vec{\Delta}_{n,j} = \frac{K_p \cdot e_j(t) + T_i \cdot \left[e_j(t) + e_j(t-1)\right] + T_i \cdot \left[e_j(t) - e_j(t-1)\right]}{threshold}$$

These coefficients determine the effective profile for the nodes, $N^* = \vec{\Delta}_n \blacklozenge N$.

The vector assignment problem as defined in Epstein, et al [14] has three pieces: items, containers, and a target function:

$$items : x^1, ..., x^n$$
$$containers : c^1, ..., c^m$$
$$function : F : \{1, ..., m\}^{\{1,...,n\}} \to \mathbb{R}^+$$

This type of assignment problem accurately represents the multi-dimensional assignment of services to nodes, where services are items, nodes are containers, and the Provisioning Norm is the target function. The items and the containers are each vectors of equal length. In Epstein, et al [14], a detailed, general framework for solving the vector assignment problem is presented. However, they make three assumptions:

1) The items are measured *a priori* off-line.

2) The items are fixed in size throughout their lifetime.

3) The items are not moved once placed.

Because our items (services) are measured on-line, assumption 1 and assumption 2 are both invalid. Further side effects of on-line measuring are: the value of the target function will change, the near-optimality is spoiled, and thus services are required to be moved. Removing these assumptions generalizes the problem in an on-line vector assignment problem. This problem is not formally addressed in the literature.

The on-line vector assignment problem is composed of items, containers, and a target function. The items are vectors whose values change over time. The containers are vectors whose values change over time. The on-line nature of the problem is the arrival of changes to the values in the vectors. While items and containers can arrive or depart, this indirectly considers the items and containers as always “present” and simply having vectors of all 0 values when they are not “present”.

Theoretical Basis

Theorem 15. The Provisioning Norm is Convex.

Proof. Prove the Provisioning Norm is Convex in $\mathbb{R}^{n \times m}$ by showing the following.

$$\left\| \lambda A + \left(1-\lambda\right) B \right\|_{\alpha,F} \leq \lambda \left\| A \right\|_{\alpha,F} + \left(1-\lambda\right) \left\| B \right\|_{\alpha,F}$$

For

$$\lambda \in [0,1]$$
$$\alpha \in [0,1]$$
$$A, B, M \in \mathbb{R}^{n \times m}$$
$$\operatorname{pos}(x) = \max\{x, 0\}$$
$$\operatorname{neg}(x) = \min\{x, 0\}$$
$$M^{+} = \left[\operatorname{pos}\left(M_{ij}\right)\right]$$
$$M^{-} = \left[\operatorname{neg}\left(M_{ij}\right)\right]$$
$$\|M\|_{\alpha,F} = (1-\alpha)\left\|M^{+}\right\|_{F} + \alpha\left\|M^{-}\right\|_{F}$$

Consider the triangular inequality of the Frobenius Norm.

$$\|A+B\|_{F} \le \|A\|_{F} + \|B\|_{F} \tag{22}$$

To this apply the pos() function defined above to each entry in each matrix.

$$\begin{aligned} \left[\operatorname{pos}\left(A_{ij}+B_{ij}\right)\right] &\le \left[\operatorname{pos}\left(A_{ij}\right)\right] + \left[\operatorname{pos}\left(B_{ij}\right)\right] \\ \left[A_{ij}+B_{ij}\right]^{+} &\le A_{ij}^{+} + B_{ij}^{+} \end{aligned} \tag{23}$$

Repeat using the neg() function defined above.

$$\begin{aligned} \left[\operatorname{neg}\left(A_{ij}+B_{ij}\right)\right] &\le \left[\operatorname{neg}\left(A_{ij}\right)\right] + \left[\operatorname{neg}\left(B_{ij}\right)\right] \\ \left[A_{ij}+B_{ij}\right]^{-} &\le A_{ij}^{-} + B_{ij}^{-} \end{aligned} \tag{24}$$

To each (23) and (24) apply the definition of $M^{+} = \left[\operatorname{pos}\left(M_{ij}\right)\right]$ and $M^{-} = \left[\operatorname{neg}\left(M_{ij}\right)\right]$.

$$\left\|[A+B]^{+}\right\|_{F} \le \left\|A^{+}\right\|_{F} + \left\|B^{+}\right\|_{F} \tag{25}$$

$$\left\|[A+B]^{-}\right\|_{F} \le \left\|A^{-}\right\|_{F} + \left\|B^{-}\right\|_{F} \tag{26}$$

The Frobenius Norm is convex giving (27).

$$\|\lambda A + (1-\lambda)B\|_{F} \le \lambda\|A\|_{F} + (1-\lambda)\|B\|_{F} \tag{27}$$

Combine (25) and (27)

$$\left\| \left[\lambda A + (1-\lambda) B \right]^{+} \right\|_F \le \left\| \lambda A^{+} \right\|_F + \left\| (1-\lambda) B^{+} \right\|_F \tag{28}$$

Rearrange (28).

$$\left\| \left[\lambda A + (1-\lambda) B \right]^{+} \right\|_F \le \lambda \left\| A^{+} \right\|_F + (1-\lambda) \left\| B^{+} \right\|_F \tag{29}$$

Similarly combine (26) and (27).

$$\left\| \left[\lambda A + (1-\lambda) B \right]^{-} \right\|_F \le \left\| \lambda A^{-} \right\|_F + \left\| (1-\lambda) B^{-} \right\|_F \tag{30}$$

Rearrange (30).

$$\left\| \left[\lambda A + (1-\lambda) B \right]^{-} \right\|_F \le \lambda \left\| A^{-} \right\|_F + (1-\lambda) \left\| B^{-} \right\|_F \tag{31}$$

Formula (30) shows the Frobenius Norm is convex over the matrix comprised of all the positive entries from the original matrix. Formula (31) shows the Frobenius Norm is convex over the matrix comprised of all the negative entries from the original matrix.

Consider the definition of the Provisioning Norm for the matrix $\lambda A + (1-\lambda) B$.

$$\begin{aligned} &\left\| \lambda A + (1-\lambda) B \right\|_{\alpha,F} = \\ &\qquad (1-\alpha) \left\| \left[\lambda A + (1-\lambda) B \right]^{+} \right\|_F + \alpha \left\| \left[\lambda A + (1-\lambda) B \right]^{-} \right\|_F \end{aligned} \tag{32}$$

Into (32) substitute (30) and (31).

$$\begin{aligned} &\left\| \lambda A + (1-\lambda) B \right\|_{\alpha,F} \le \\ &\qquad (1-\alpha) \left[\lambda \left\| A^{+} \right\|_F + (1-\lambda) \left\| B^{+} \right\|_F \right] + \alpha \left[\lambda \left\| A^{-} \right\|_F + (1-\lambda) \left\| B^{-} \right\|_F \right] \end{aligned} \tag{33}$$

Apply distribution of multiplication over addition.

$$\begin{aligned} &\left\| \lambda A + (1-\lambda) B \right\|_{\alpha,F} \le \\ &\qquad (1-\alpha) \lambda \left\| A^{+} \right\|_F + (1-\alpha)(1-\lambda) \left\| B^{+} \right\|_F + \alpha\lambda \left\| A^{-} \right\|_F + \alpha (1-\lambda) \left\| B^{-} \right\|_F \end{aligned} \tag{32}$$

Rearrange.

$$\begin{aligned}&\left\|\lambda A+(1-\lambda)B\right\|_{\alpha,F}\le\\&\quad\lambda\left[(1-\alpha)\left\|A^{+}\right\|_{F}+\alpha\left\|A^{-}\right\|_{F}\right]+(1-\lambda)\left[(1-\alpha)\left\|B^{+}\right\|_{F}+\alpha\left\|B^{-}\right\|_{F}\right]\end{aligned}\tag{33}$$

Use $\left\|M\right\|_{\alpha,F}=(1-\alpha)\left\|M^{+}\right\|_{F}+\alpha\left\|M^{-}\right\|_{F}$ for substitution in for *A* and *B* (33).

$$\left\|\lambda A+(1-\lambda)B\right\|_{\alpha,F}\le\lambda\left\|A\right\|_{\alpha,F}+(1-\lambda)\left\|B\right\|_{\alpha,F}\tag{34}$$

The next few theorems present the lower and upper bounds of the Provisioning Norm. Each bound is expressed in terms of number of services, *s*, number of nodes, *n*, number of resources, *r*, the parameter, α, and assumed residual resource values, *ρ*. The matrix *N* is *n* by *r* representing the nodes' resources. The matrix *S* is *s* by *r* representing the services' resources. The adjacency matrix *C* is *n* by *s*, assigning services to nodes. A service can only be assigned to one node. The following constraints hold for matrix entries.

$$\begin{aligned}&N_{ij}\in[0,1]\\&S_{ij}\in[0,1]\\&C_{ij}\in\{0,1\}\end{aligned}$$

The nodes' residual resources are expressed by *N-CS*. These values are directly responsible for the value of the Provisioning Norm when applied by assigning services to nodes. Low values represent better assignments.

Definition 23. The Frobenius Norm (equivalent to the vector 2-norm) is lower and upper bounded by the 1-norm as defined for the matrix *A* with *a* entries by $\frac{1}{a}\left\|A\right\|_{1}\le\left\|A\right\|_{F}\le\left\|A\right\|_{1}$.

Theorem 16. The lower and upper bounds of the Frobenius Norm of the residual matrix is

$$\frac{1}{\sqrt{nr}}\left\|N-CS\right\|_1 \leq \left\|N-CS\right\|_F \leq \left\|N-CS\right\|_1 .$$

Proof. Given Definition 23 and the fact there are *nr* entries in the residual matrix the lower bound of the Frobenius Norm is $\frac{1}{\sqrt{nr}}\left\|N-CS\right\|_1 \leq \left\|N-CS\right\|_F \leq \left\|N-CS\right\|_1$.

Theorem 17. The Provisioning Norm lower and upper bounds of the residual matrix *N-CS* is $\frac{1}{\sqrt{nr}}\left((1-\alpha)\left\|N\right\|_1-\alpha\left\|S\right\|_1\right) \leq \left\|N-CS\right\|_{\alpha,F} \leq \left((1-\alpha)\left\|N\right\|_1-\alpha\left\|S\right\|_1\right)$.

Proof. Consider the lower bound first. In this case, the lowest value the norm takes depends on the assignment of the services to the nodes. The 'tightest' assignment of services possible is the minimum value. Depending on the value of this could be the assignment that minimizes under-provisioning or minimizes over-provisioning.

Let $M = N - CS$ be the residual matrix for abbreviated notation. Consider the Provisioning Norm $\left\|M\right\|_{\alpha,F} = \left[(1-\alpha)\left\|M_+\right\|_F + \alpha\left\|M_-\right\|_F\right]$. Use the lower bound of the Frobenius Norm to get a lower bound,

$$\left[\frac{(1-\alpha)}{\sqrt{nr}}\left\|M_+\right\|_1 + \frac{\alpha}{\sqrt{nr}}\left\|M_-\right\|_1\right] \leq \left[(1-\alpha)\left\|M_+\right\|_F + \alpha\left\|M_-\right\|_F\right]$$

Case I. If $\left\|N\right\|_1 - \left\|S\right\|_1 > 0$ then there is theoretically enough resources for all services to be assigned with left over room. This also indicates there are no negative entries in *N-CS*. Drop the second term in the lower bound because $M_- = \mathbf{0}$.

$$\frac{(1-\alpha)}{\sqrt{nr}}\left\|M_+\right\|_1 \leq \left[(1-\alpha)\left\|M_+\right\|_F + \alpha\left\|M_-\right\|_F\right]$$

Note if all services are assigned then $\left\|CS\right\|_1 = \left\|S\right\|_1$ and then

$$\begin{aligned}\|M_+\|_1 &= \|N - CS\|_1 \\ &= \|N\|_1 - \|CS\|_1 \ . \\ &= \|N\|_1 - \|S\|_1\end{aligned}$$

Substituting yields:

$$\frac{(1-\alpha)}{\sqrt{nr}}\left(\|N\|_1 - \|S\|_1\right) \le \left[(1-\alpha)\|M_+\|_F + \alpha\|M_-\|_F\right].$$

Case II. If $\|N\|_1 - \|S\|_1 < 0$ then there is more demand for resources than available resources. In this case, the best theoretical configuration to minimize the Provisioning Norm is an assignment such that all resources are consumed exactly and any extra services are excluded. Thus for $M = N - CS$, $M_+ = \mathbf{0}$ and $M_- = \mathbf{0}$.

The upper bound has two cases as well. One case has no services assigned to any node. The second case has all services assigned to the smallest node.

Case I. If no services are assigned then $N - CS = N - \mathbf{0}S = N$. As above using the upper bounds of the Frobenius Norm, substitution yields,

$$\left[(1-\alpha)\|M_+\|_F + \alpha\|M_-\|_F\right] \le \left[(1-\alpha)\|M_+\|_1 + (1-\alpha)\|M_-\|_1\right]$$

because $N - CS = N - \mathbf{0}S = N$, there are only non-negative entries; yielding

$$\left[(1-\alpha)\|M_+\|_F + \alpha\|M_-\|_F\right] \le (1-\alpha)\|N\|_1 .$$

Case II. All service could be assigned to the smallest of all the nodes. Assume the smallest node has zero resources. The positive entries of *N-CS* then is N and thus the 1-norm of the positive residual matrix is equal to the 1-norm of N. Correspondingly the negative entries of *N-CS* is all zeros except the row to which all of the services are assigned and thus the 1-norm of the negative entries

$$\left\|\left[N - CS\right]_+\right\|_1 = \left\|N\right\|_1$$
$$\left\|\left[N - CS\right]_-\right\|_1 = \left\|S\right\|_1$$

Theorem 18. If a service placed on a node changes size sufficiently, the configuration is no longer locally optimal.

Proof. For a node vector *n*, assume the *m* services assigned to it do not in sum consume more resources than the node offers, $\forall i, n_i > \sum_j^m s_{j,i}$. Further assume the current assignment for this node is locally optimal.

If *n* has more than one service (*m*>1) assigned to it, setting one service s_1 equal to the size of the node

$$\begin{array}{l} \forall i, s_{*,i} \leftarrow n_i \\ s_* = s_1 + s_\delta \end{array}.$$

Under this new condition, dropping any (or all) of the other services yields a more optimal arrangement for this node. From the global perspective, it may be more optimal to move the enlarged service *s₁* to another node. Regardless, the current configuration under the new condition of the enlarged service is no longer locally optimal.

If *n* has one service (*m*=1) assigned to it, set the service s_1 to something larger than the size of the node

$$\begin{array}{l} \forall i, s_{*,i} \leftarrow n_i + \delta \\ s_* = s_1 + s_\delta \end{array}.$$

Under this new condition, dropping the other yields a more optimal arrangement for this node. From the global perspective, it may be possible with a low α and a low δ

that dropping the service is not more optimal; δ may be large. Regardless, the current configuration under the new condition of the enlarged service is no longer locally optimal.

This theorem provides the existence of an amount the service can grow to force a configuration change. The lower bound of this value s_{δ}^{*} is a complicated function of the individual node and service profiles, number of nodes and services, their respective sizes, and the settings of policy parameters. It is expected this lower bound is much less than the size of the node, $0 < s_{\delta}^{*} < n_{i}$.

Because the service resource profiles are a function of the number of requests per second the service is servicing, the increase in the size of the service profile s_{δ} may be due to an increase of demand for the service. This implies as the consumption of the service increases the profile grows and changes to the configuration occur. This has been shown previously.

Alternatively as mentioned, the profile can be artificially increased to invoke a move. The question becomes how much to increase the profile and on what that increase is based. The goal is to improve performance in terms of response time. A feedback control loop from control theory using the past performance as an input provides a real possibility.

The feedback control requires inputs: the measured performance and the target performance. These two values determine the error which drives the feedback. The measured performance is easily attained by capturing the response times. The selection of the target performance is a matter of policy. Furthermore, the model presented here ignores feedback values less than one. If values less than one are allowed, then as a service's response time drifts lower away from the target, the service's profile would be decreased accordingly. This would imply the service needs fewer resources than measured. The implication of this approach may have merit and would mathematically drive all service response times to the target. However, there is the possibility that a smaller number of services in a resource rich environment might perform very well regardless of the size of its profile and could then be moved to a smaller server where performance would suffer. For this work, the target is a threshold, any response time below it is not subject to control, and the coefficient is set to 1.

The model defined earlier is

$$K_p = 0.6$$
$$T_i = 0.5$$
$$T_d = 0.125 \qquad .$$

$$\vec{\Delta}_{s,i} = \frac{K_p \cdot e_i\left(t\right) + T_i \cdot \left[e_i\left(t\right) + e_i\left(t-1\right)\right] + T_i \cdot \left[e_i\left(t\right) - e_i\left(t-1\right)\right]}{threshold}$$

Putting the threshold in the denominator normalizes the feedback value relative to the threshold. This makes the feedback more sensitive to bad performance with smaller thresholds, small errors can still yield sufficient feedback. Further consider the size of the error and the feedback $\vec{\Delta}_{s,i}$. Small errors will yield small feedback. In this case, the service is above the threshold and is being adjusted, but the adjustment may not be enough to invoke a change:

$$\vec{\Delta}_{s,i} \cdot s_i - s_i < s_\delta^* .$$

Under these conditions, performance suffers and nothing is done about it. This implies the mean performance for a service under control could settle to a response time above the threshold and thus, in the strictest sense, the threshold needs to be below the truly desired service level agreement.

Theorem 19. In the vector packing assignment problem, given the set of bin vectors and two sets of items, if one set of items is sufficiently larger than items in other set, the first set has fewer possible acceptable assignments.

Proof. Let *N* be a set of bin vectors. Let *S* be a set of packing vectors. If all individual vectors in *S* are each larger than each vector in *N*, then the size of the set of acceptable configurations *C* is **0**; $\forall \vec{s} \in S, \forall \vec{n} \in N, \vec{s} > \vec{n} \Rightarrow |C| = 0$. If there is one and only one vector $\vec{s}$ in *S* and one and only one vector *n* in *N* such that $\vec{s} < \vec{n}$, then there is only one acceptable assignment $\exists ! \vec{s} \in S, \exists ! \vec{n} \in N, \vec{s} < \vec{n} \Rightarrow |C| = 1$. As a second vector $\vec{t}$ in *S* becomes smaller than a vector $\vec{n}$ in *N*, there are two possible assignments, either $\vec{s}$ is assigned to $\vec{n}$ or $\vec{t}$ is assigned to $\vec{n}$; $\exists ! \vec{s}, \vec{t} \in S, \exists ! \vec{n} \in N, \vec{s} \neq \vec{t}, (\vec{s} < \vec{n}) \wedge (\vec{t} < \vec{n}) \Rightarrow |C| = 2$. The cardinality of *C* grows until the sum of all vectors from *S* is smaller than all vectors in *N*, meaning any collection of packing vectors can be assigned to any bin vector. Under these conditions the cardinality of *C* is the number of bin vectors raised to the number of packing vectors; $\forall \vec{n} \in N, \sum_{i=1}^{|S|} S_i < \vec{n} \Rightarrow |C| = |N|^{|S|}$. Therefore, as packing vectors become sufficiently smaller the set of acceptable assignments grows larger, and as packing vectors become sufficiently larger the set of acceptable assignments gets smaller.

Theorem 20. Applying control to the service resource profile coefficients will have better performance than applying no control.

Proof. Consider the performance of a single service *s* and $\vec{s}$ be its resource vector. Let *perf(s)* be the response time of a service s. Let *threshold* be an acceptable response time. If *perf(s)* < *threshold*, then nothing happens. If *perf(s)* > *threshold*, then control is invoked. Control is invoked by setting the corresponding scalar entry for this service in the service adjustment vector $\vec{\Delta}_s$ (Definition 18) to something greater than one, let δ be this scalar value and $\vec{s}^* = \delta \cdot \vec{s}$. Let *n* be the node to which *s* is assigned and $\vec{n}$ be the resource vector for the node. Let $S^{(n)}$ be the set of all service vectors assigned to node *n*. Let $\forall j, \vec{v}_j = \sum_i S_{ij}^{(n)}$ be the sum of all resource vectors assigned to node *n*. If $\vec{v} - \vec{s} + \vec{s}^* < n$, then increasing the size of *s* was insufficient to cause certain change, e.g. the service still fits. No moves are guaranteed. This likely means either the profiling process is not accurate or the *threshold* value is too low. If $\vec{v} - \vec{s} + \vec{s}^* > n$ then one of three possibilities occur. The service *s* is (1) dropped from node *n* and not assigned to any other node. In this case, the service performance is null and no longer included in the overall service performance mean. Therefore, performance will improve because *s* was performing above the threshold. The service s could be (2) moved from node *n* to another. In this case the $\vec{s}^*$ mathematically fits on the new node. Because $\vec{s}^* > \vec{s}$, the service is being given more resources than it actually needs on the new node. It is conjectured that given more resources than required, performance will increase and, therefore, moving the service *s* will increase performance. If the service *s* is not moved and other services exist on node *n*, $\vec{v} - \vec{s} > 0$, then (3) one of the other services will be dropped. In this case, as in the previous case, the service is provided more resources, $\vec{s}^*$, than it actually needs, $\vec{s}$, on the current node. The other (moved) service is no longer consuming resources on node *n* making them available to the other services on the node, including *s*. In all three cases performance improves.

Theorem 21. Given Theorem 19, the local minima configurations remaining after increasing the size of services (items) are the better performing configurations.

Proof. First consider the following informal discussion utilizing a Venn diagram. In Figure 28, the circle containing A represents the set of local minima configurations the greedy heuristic and quality function produce given the fixed node

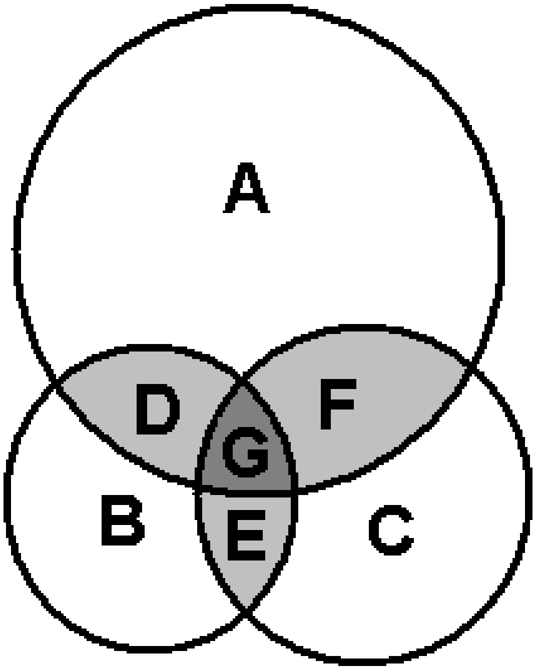


Figure 28. Intersection of Configurations Found Without Control (A), Configurations Found With Control (B), and Configurations Which Perform Well (C).

resource profiles *N* and the fixed *actual* service resource profile *S*, e.g. quality fits. The circle containing B represents the configurations where the performance is acceptable. The circle containing C represents the local minima configurations which the greedy heuristic and quality function determine; given the fixed node resource profiles *N* and the *effective* service resource profiles S^*.

The initial configuration (after services have been reasonably profiled) selected by the greedy heuristic could be in one of two regions, A or D+G. If it is in A then performance will not be acceptable and control will be invoked. If it is in D+G then performance will be acceptable and no control will be invoked. As control is invoked and the greedy search starts with the *current configuration* for its search, the new configuration will be in the circle C+E+F+G where the effective service resource profile local minima are located. If the configuration is in C, control will continue and this circle will move (contain different configurations based on the effective profile). If the configuration is in G+E, performance is acceptable and the effective profile will return to the actual profile. If it is in G, performance and fit are acceptable and nothing will change after this until conditions change. If it is in E, once the effective profile returns to the actual profile, the greedy heuristic will engage again using the current configuration as the start and hopefully will end up in D+G. If not, then the process starts over in a converging fashion, e.g. the configuration will be closer to the well performing set than before. This is the basis of control theory. Closer is defined as the Hamming distance from the current configuration to any of the configurations in D+G is less than it was before.

More formally consider the set of all possible configurations. Given *constant* matrices *N*, *S*, and the quality function *q()*. Each configuration is assigned a positive, real-valued quality, $q_{N,S} : C \to \mathbb{R}^{+}$, the heuristic uses *q()* to find the local minima. Consider an alternative function *p()* mapping each configuration to a vector of performance values, $p_{N,S} : C \to \mathbb{R}^{+}_{1\times s}$, where *s* is the number of services. For the purposes of this proof, the assumption is that each configuration *C* has a unique performance. The assumption means in the real world fixing *N*, *S*, and hits per second on each service that implementing a particular configuration will yield the exact performance each time.

For each configuration the *p()* values are used to change the actual profile *S* to the effective profile $S^{*} = p_{N,S}\left(C\right)\blacklozenge S$. If quality *q()* is changed to include performance *p()* then $q_{N,S} : C, p_{N,S}\left(C\right) \to \mathbb{R}^{+}$. The Provisioning Norm becomes $\left\| N - C\left(p_{N,S}\left(C\right)\blacklozenge S\right)\right\|_{\alpha,F}$. Assume there are a local minimum *C* and its neighbor C^{*}, $q_{N,S}\left(C\right) < q_{N,S}\left(C^{*}\right)$. Neighbor is two configurations with a Hamming Distance of less than or equal to 2, $hamm\left(C,C^{*}\right) \leq 2$. Each configuration has a respective performance, $p_{N,S}\left(C\right), p_{N,S}\left(C^{*}\right)$.

Consider vector of positive scalars *δ* that can be added to *p()*. Further consider the effect it has on the quality function. There exists such a *δ* where the adjusted quality function is greater than the unadjusted quality function.

$$\begin{aligned}
&\exists \delta, \delta \in \mathbb{R}^{+}_{1\times s} \\
&p_{N,S}\left(C\right) < p_{N,S}\left(C\right) + \delta \\
&\left\| N - C\left(p_{N,S}\left(C\right)\blacklozenge S\right)\right\|_{\alpha,F} < \left\| N - C\left(\left[p_{N,S}\left(C\right) + \delta\right]\blacklozenge S\right)\right\|_{\alpha,F}
\end{aligned}$$

Further consider if the neighbor's whose performance is such that it sufficiently effect quality so the current configuration is no longer the local minimum.

$$p_{N,S}\left(C^{*}\right) \geq p_{N,S}\left(C\right) + \delta \Rightarrow q_{N,S}\left(C^{*}\right) < q_{N,S}\left(C\right)$$

In this, the local minimum has moved *and* moved to a configuration of better performance. Because of Theorem 19, there are less minima because the size of the services S^{*} are sufficiently larger than *S*. The minima remaining must therefore be of better quality. Because the heuristic is greedy it will take advantage of the lower quality value and select these performing configurations.

Unfortunately *p()* is performance and determining *p()* a priori is a complex endeavor. The values of *p()* in a real-time testbed can only be sampled one point at a time, e.g. the current configuration. Given Theorem 21, performance should improve applying a performance based controller.

Empirical Results

The empirical inquiry involves 25 web servers (nodes) and 100 services. Each service is exercised from a predefined load. The predefined load was generated from a Poisson process using Knuth’s method [82] with a mean of 5. Each run of the predefined load is a trial. Four cases are considered in this work. The first case applies no control. The following three cases progressively adjust a control parameter. The control parameter can loosely be interpreted as a target response time. Each case spans a set of 25 trials.

The predefined load lasts 40 minutes. The first five minutes allows for the nodes to reboot and organize themselves into their communication topology. The second five minutes creates the services and allows the nodes to determine a configuration and assign the services to their respective nodes. Note: the services’ initial profiles are $\varepsilon=2^{-10}$ (essentially zero) because no requests are occurring and because a service with a profile of all zeros is never mathematically added to a node due to the greedy nature of the heuristic. At the ten minute mark, the load of traffic with mean of 5 requests per second begins and continues for 30 minutes. Finally, the nodes and *loadrunners* reboot themselves and repeat the process.

Because the services are not profiled, the initial arrangement of services is essentially arbitrary due to the fact each service is the same size. After the 5 requests per minute arrive, services are profiled. These profiles are communicated among the nodes, which begin to find better arrangements. Each new configuration takes approximately 60 seconds to be found, disseminated, and implemented. At the beginning of each trial, the arbitrary assignments lead to substantially high response times. This bootstrapping time frame is ignored in calculating the statistics below.

The tables below utilize the Wilcoxon rank-sum test [101] (also called the Mann-Whitney U test) as a significance test to demonstrate that the difference between the cases is statistically strong. The response times (and other data below) are not statistically normal. Each is heavy tailed. The rank-sum test is a multiple step process. First, the two sets of considered data are combined into a single set of data. The data are sorted from lowest to highest. Each is assigned its ordinal rank. The mean and standard deviation of the ranks are calculated. These values are used to determine a p-value which can be used to accept or reject a hypothesis with a prescribed confidence (99.5%). The p-values are calculated using MATLAB's *ranksum()* method using an α=0.005. For response time, the p-values were all very near zero and thus the conclusion can be made that controlling the coefficient clearly helps reduce the response time.

The response time over all services for each case is presented in Table 15. The first column specifies the target value used in the control. The second column is the average service response time after t=1800. Before 1800 the service profiles and configurations are settling from the introduction of requests on all 100 services. As seen in later graphs of response time, the initial response times spike as proper profiling and configurations are set. The third column shows the mean response times for the last 100 seconds of each case. The control has reduced the response times.

Table 15. Response Times with Varying Levels of Control.

Service Control	Response Time (ms) *t*>1800	Response Time (ms) *t*>2300
-	77.635	71.135
100ms	32.864	25.866
50ms	24.607	20.234
10ms	30.157	14.750

Table 16 shows how each case included services after t=1800. Including a service means the service is assigned to a node and is providing responses to the requests. The Provisioning Norm parameter α is set to 0.99999 for all cases. This demands services to be excluded if they do not fit on any node. The data in Table 16 shows that controlling the coefficient *over time* does not reduce the number of services included; the difference in these values are not statistically relevant. The statistical Wilcoxon rank sum test gave p-values indicating there was not sufficient difference to draw the conclusion the means were significantly different.

Table 16 - Services Included with Varying Levels of Control (t>1800)

Service Control	Services Included
-	99.495
100ms	99.543
50ms	99.801
10ms	99.195

Table 17 shows the amount of services moved in terms of the Hamming distance between each configuration. Recall, the configuration is represented as a 0,1 adjacency matrix mapping services to nodes. If a service is moved, it is counted as 2 moves, a drop and an addition. The Hamming distance between two configurations, A and B, is

$$\sum_{ij} \left| a_{ij} - b_{ij} \right|$$

A significant amount of additional moves were not generated when controlling the coefficient even though controlling it forces moves to occur. This implies in the same number of moves with no control, the control yields a well performing configuration. This should be investigated further in order to adequately substantiate this claim with more statistical rigor.

Table 17 –Hamming Distance with Varying Levels of Control (t>1800)

Service Control	Moves
-	8.092
100ms	8.657
50ms	8.694
10ms	6.310

Table 18 shows the average control coefficient for each case after t=1800 and for coefficients greater than 1. This is expected based upon the design of the controller. Lower thresholds will be further from the actual performance creating larger error. Furthermore, the control model has the threshold in the denominator and thus a lower value yields a larger value.

Table 18 – Coefficient Values with Varying Levels of Control (t>1800 and coefficient>1)

Service Control	Service Coefficient
-	-
100ms	7.393
50ms	11.924
10ms	114.618

Table 19 and Table 20 show the average queue length and corresponding response times. Table 19 shows for queues of size greater than zero and after t=1800 the mean queue size and response time are reduced with control. The means are significantly different using the Wilcoxon rank sum statistic with 99.5% confidence. Controlling the coefficient indirectly reduces the size of the queues. Table 20 shows for empty queues the response times are also improved as a result of controlling the coefficient.

Table 19 – Queue Size with Varying Levels of Control (size>0 and t>1800)

Service Control	Mean Queue Size	Mean Response Time (ms)
-	6.365	529.647
100ms	2.333	193.423
50ms	2.293	156.315
10ms	4.110	227.155

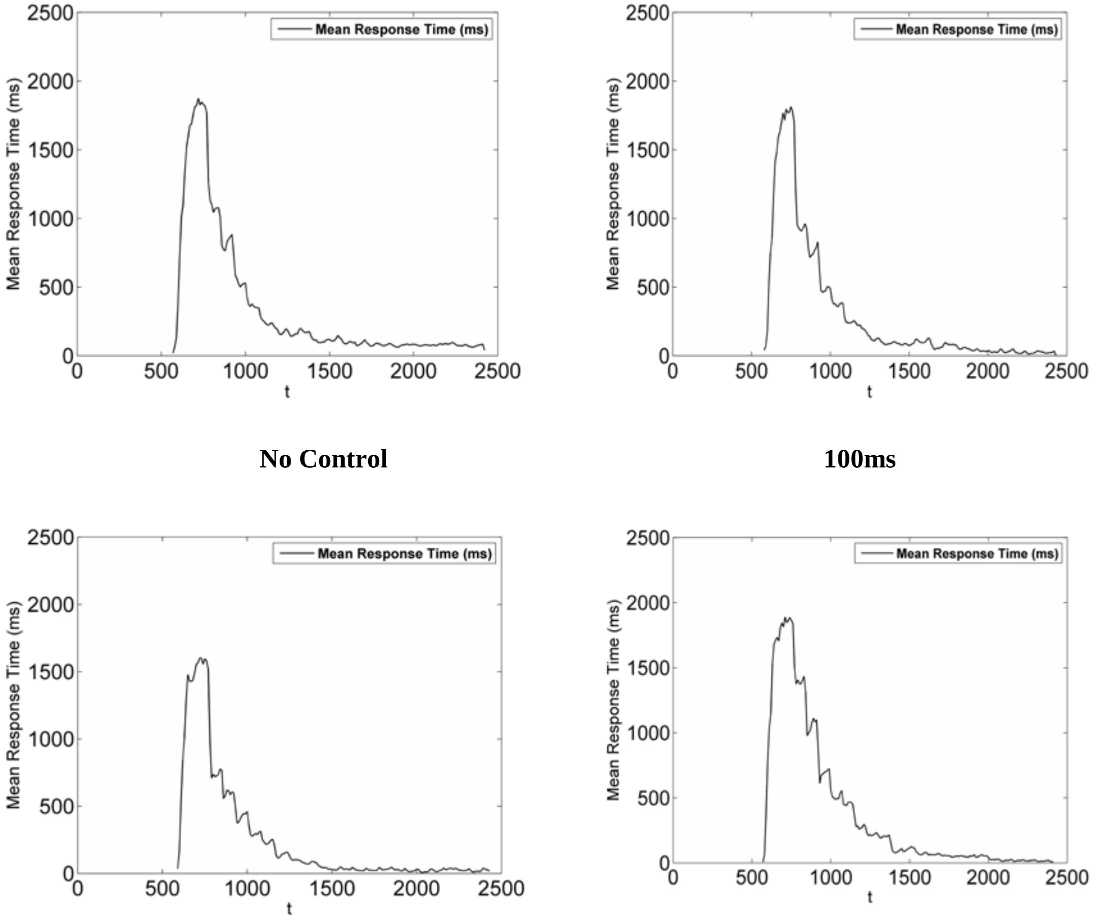
Table 20 – Queue Size with Varying Levels of Control (size=0 and t>1800)

Service Control	Mean Queue Size	Mean Response Time (ms)
-	0	14.188
100ms	0	14.261
50ms	0	11.486
10ms	0	8.611

Figure 29 shows the mean response time of all services across time for each case. Once the traffic begins the service response times suffer substantially due to the services having been placed using profiles of essentially zero. As the traffic continues to arrive the profiles become more representative of reality. Better configurations are applied until response times settle, on average, to below the threshold. The no control case however does not settle as low as the others; refer to Table 15 for specific values.

Figure 30 shows the number of services included over time for each case. In the three control cases once traffic arrives, approximately t=600, the control causes service profiles to increase such that services are dropped and not added until the system becomes more stabile. In all cases the number of services included returns to near 100.

Figure 31 shows the mean value of the controlled coefficient of all services across time. Clearly the 10ms case shows much higher values. In all three cases the value is increased once traffic arrives and then settles back to 1.



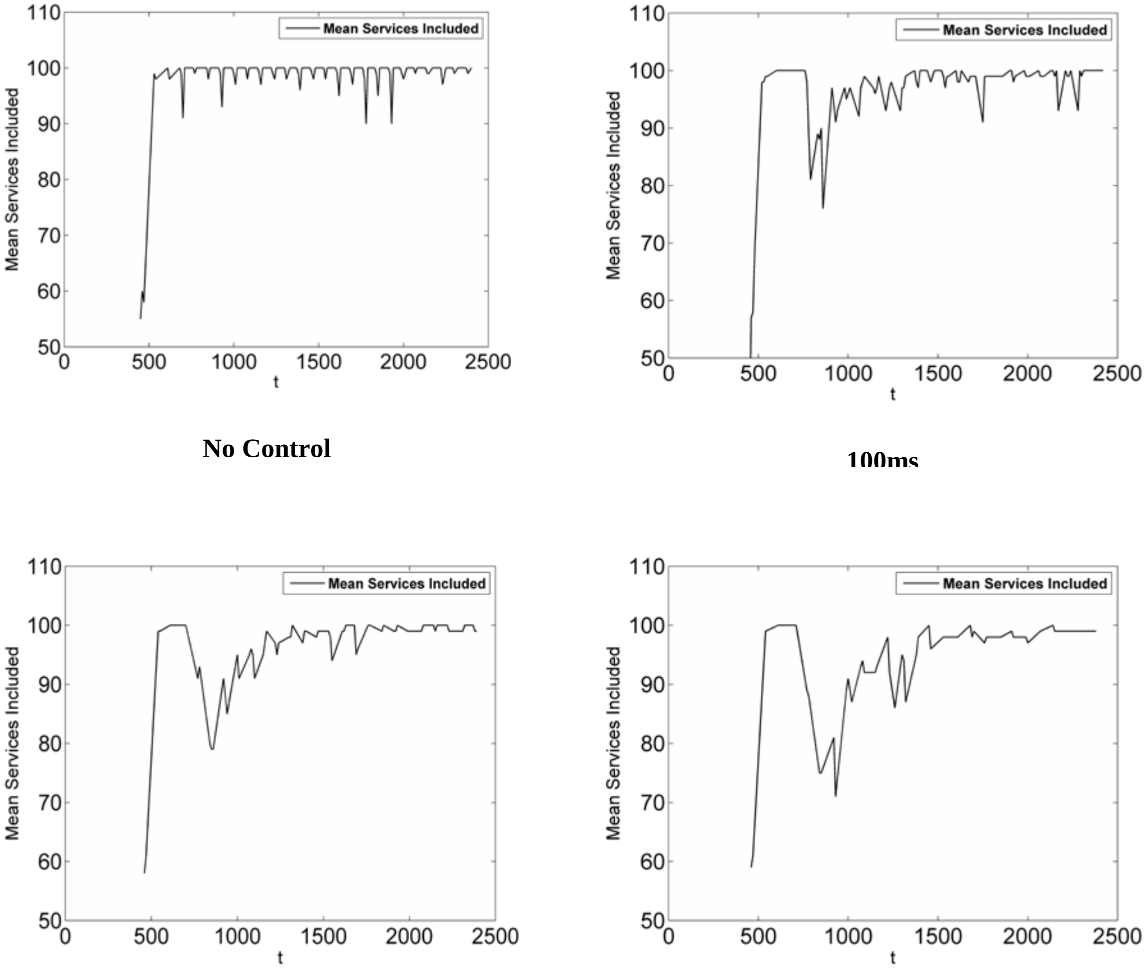
No Control

100ms

50ms

10ms

Figure 29. Mean Response Times Over Time.



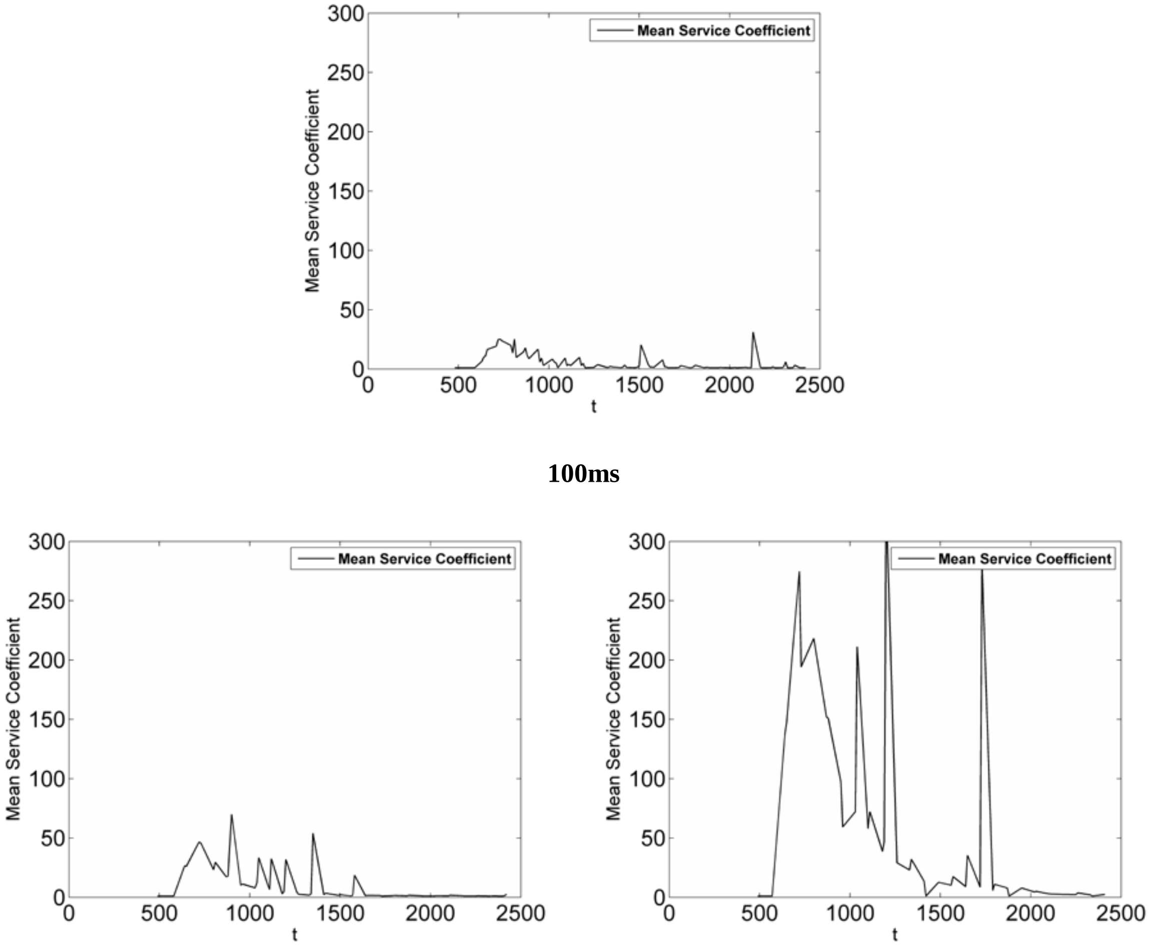
No Control

100ms

50

10

Figure 30. Mean Services Included Over Time.



100ms

50ms **10ms**

Figure 31. Coefficients Over Time.

Summary

This work presents the effects of applying control theory to a large number of co-operative, self-organizing web servers. The performance of individual web services was improved using a Proportional Integral Derivative control feedback loop based on the response time error. The control increases the effective size of each service's resource profile making the service require more resources as performance decreases. The changes to the service profiles lead to services being moved from an under-performing configuration to a well-performing configuration. This was demonstrated theoretically and empirically. Theoretically the problem was presented as an on-line dynamic vector packing problem. The theorems demonstrate the number of acceptable configurations decreases as the size of the items increase and conditioning poor performing service profiles mathematically eliminates local minima having poor performance. Reflecting the theoretical basis, the empirical results showed when control is applied the service response times and queue sizes improve up to 66%.

Future works based on this work can include examining controlling the node resource profile by reducing the resources it makes available when performance is suffering on the server. Additionally, exploring the control parameters beyond the assumed Zeigler-Nichols tuning is open to investigation. Applying a control to the parameters is also an open topic. Finally, investigating how control fares as traffic varies widely over time will provide interesting problems.

This work contributes an improved technique for placing services on servers such that performance and fit drive the arrangement. This work provides administrators additional flexibility in managing a large number of services by providing mathematical parameters that yield predictable outcomes to meet policy goals.

V. Conclusions and Recommendations

This dissertation researches the mathematical translation of resource provisioning policy into mathematical terms and parameters. In ever larger and larger scale computational environments, the automation of system management becomes increasingly important. The automation of the system management will require computers calculating the conditions of system mathematically evaluating the context of these conditionals, and numerically construct actions. This work investigates specifically the web service placement problem, the performance of web services, and the mathematical policies created to yield specific results.

This work brings together several approaches in a unified solution to the On-Line Service Placement Problem. No previous work incorporates on-line service profiling, service re-assignment, decentralized heuristic search, self-organization, performance feedback, and asymmetric measures. The previous works such as [4] [6] [7] [14] [39] [63] [78] [89] each address one of these aspects in isolation and fail to consider the interactions and effects each have on the others.

Chapter 2 outlined the layered framework building from the physical, quantitative, qualitative, to the policy layer. This framework abstracts values from conditions and conditions from actions allowing mechanisms to change while policies remain the same. Chapter 2 described the nature of on-line and metrical problems demonstrating the placement problem is on-line, metrical, vector packing problem. In this setting, at large scale, random algorithms outperform deterministic. All of this sets up the problem in five parts: measurement of resource usage, measurement quality of placements, measurement of performance, parameters for placement and performance tolerances, and efficiently determining the new placements with a randomized algorithm.

This dissertation frames and solves a problem not yet addresses in the literature, a generalized vector packing problem. The problem addressed here allows for items that have been placed to change in size, and to be moved as needed in response to these changes and in response to newly arrived items. Additionally this problem has items that are multi-dimensional and the solutions are discovered in a decentralized fashion. Each of these has been discussed in the literature; however, these were additional constraints making the problem more challenging.

Chapter 3 described the Cloud Chamber, the testbed used to gather real-time performance data in response to traffic loads placed on web services. These services were arranged on the servers in response to the policies and parameters set forth by each experiment. This was a flexible self-organizing environment allowing for the theoretical to be verified with empirical results.

Chapter 4 introduces the Provisioning Norm. The Provisioning Norm combined with the random heuristic proves to be theoretically effective but also computationally efficient. The Provisioning Norm is effective at separating good placement from bad placements by creating a partial ordering of placements biased with a policy parameter. The policy parameter allows for the preference of inclusion: the inclusion of all services at the expense of degraded performance or the acceptance of exclusion with the expectation of improved performance. Theorem 3 and Theorem 4 proves the ordering, separation, and point of separation. The greedy heuristic finds a placement in a number of rounds less than the number of services. The empirical data in Chapter 3 demonstrate employing numerical methods allows one to determine the policy parameter value to include a specific number of services. The empirical data also supported the change in performance as the parameter changed the inclusion. Further discussion and proofs show why the Provisioning Norm is a better choice than other matrix and vector norms and of the Kullback-Leibler function.

Chapter 5 demonstrates with dynamic profiling and changes to traffic, over time static placements do not perform well. Services must be moved in response to changes in traffic loads. This simple experiment and demonstration leads to the question of caching configurations. If the system were to return to a previous state does caching save time or improve performance? The conclusion was that performance was not improved with caching. Also, new and better configurations could be found very quickly and caching was not useful in this regard. However, caching recently used configurations for starting points in the greedy search heuristic did substantially minimize the cost of changing (service moves) to the new configuration from the old. To re-iterate, starting a greedy search from where one is at finds the nearest local minima of quality. However, starting from random locations lower local minima are found but are further away from the current location, in other words a “thinking out of the box” policy non-cached starting locations.

Chapter 6 incorporated performance feedback into the mathematical decision making of the placement selection. Increasing service profile size improves performance was proven. Empirical verification for this theory was presented. The idea presented here is that the profiling mechanism may not accurately reflect the situation on each server. The increase in the profile size is conjectured to be the additional overhead (memory page swapping, disk contention, etc) experienced by the service at that moment. Supporting proofs, proved along the way include the upper bounds, lower bounds, and convexity of the Provisioning Norm. Also in general vector packing problems, the larger the items are the fewer acceptable solutions are available. The empirical data also demonstrated that using the performance control feedback to swell the services indirectly decreased the size of the queues in the web servers substantially.

This work opens several research topics for future work. Empirical results for larger scale environments are the next logical step for the testbed. Leasing time on a virtualized platform such as Amazon or a university cloud would provide the ability to verify these techniques with hundreds of nodes and thousands of services. The most significant challenge for the scale is the matrix operations embedded in the Provisioning Norm. It is conjectured that using some sparse matrix operations could improve this performance as scale exponentially grows. A further consideration in scaling up is to federate pools of nodes into manageable size clusters. The Provisioning Norm could be used at the federated level where a node represents an aggregation of the pool's resources and a service could represent an aggregation of a bundle of services.

The model presented in this work assumed a decentralized self-organization. This prevents the need for a centralized authority and the processes of finding and electing a leader. However this comes at the expense of time required to gain consensus. Research into the costs and benefits of each can be pursued, particularly at large scales and for network partitions.

This work assumed the equal priority/cost/profit of services and nodes. The mathematical integration of priority is an essential next step. One consideration was to make the priority, cost, and profit or these items as columns in the service and node matrices, e.g. resources. Further possibilities include implementing priority/cost/profit in the heuristic as it chooses which services to drop and add.

Using control feedback loops the performance of each service affected each service's profile size. The size was increased as response times increased. In future work, the size of the node profile should be reduced as either the collective response times of the services housed on the node suffer or possibly in response to the internal operating system measures like CPU utilization, etc. Further consideration should be given to the tuning of the PID controllers. In other works, controllers were added to control the tuning parameters of the controllers, e.g. controlling controllers. This additional layer of controllers could be applied to the individual nodes or possibly act system wide.

Overall, this work translates performance goals and policies into mathematical parameters. The translation is both theoretically proven and empirically verified. Specifically, as response times suffer due to external pressures, such as increased traffic volume or decreased available resources, the system autonomously, per policy directives, excludes services, searches for extremely different placements, and increases service profiles to press for better placements. These autonomous abilities are required for large scale system administration and are not in the current literature.

VIII. Bibliography

[1] M. Chang, J. He, E. Castro-Leon, "Service-Orientation in the Computing Infrastructure," *Proceedings of the Second IEEE international Symposium on Service-Oriented System Engineering*, pp. 27-33, 2006.

[2] J. Kephart, "Research challenges of autonomic computing," *Proceedings of the 27th international Conference on Software Engineering*, ICSE '05, pp. 15-22.

[3] T.D. Braun, H.J. Siegel, N. Beck, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," Journal of Parallel and Distributed Computing, vol. 61, pp.810-837, 2001

[4] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic Placement for Clustered Web Applications," *Proceedings of the 15th international conference on World Wide Web*, WWW2006, pp. 595-604, 2006.

[5] M. Steinder, I. Whalley, and D. Chess, Server virtualization in autonomic management of heterogeneous workloads. *IFIP/IEEE Sym. on Integrated Network Management*, Munich, Germany, 2007.

[6] B. Urgaonkar, A. Rosenberg, P. Sheony, "Application Placement on a Cluster of Servers," *International Journal of Foundations of Computer Science*, vol. 18, pp. 1023-1041, 2007.

[7] Y. Zhang, Z. Wang, B. Gao, C. Guo, W. Sun, X. Li, "An Effective Heuristic for On-Line Tenant Placement Problem in SaaS," pp. 425-432, *2010 IEEE International Conference on Web Services*, 2010.

[8] B. Speitkamp, M. Bichler, "A Mathematical Programming Approach for Server Consolidation Problems in Virtualized Data Centers," *IEEE Transactions on Services Computing*, vol.3, no.4, pp.266-278, Oct.-Dec. 2010

[9] I. Cunha, J. Almeida, V. Almeida, M. Santos, Self-Adaptive Capacity Management for Multi-Tier Virtualized Environments, *10th IFIP/IEEE International Symposium on Integrated Network Management*, IM '07, pp.129-138, 2007.

[10] X. Zhu, D. Young, B.J. Watson, Z. Wang, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, 1000 islands: an integrated approach to resource management for virtualized data centers, *Cluster Computing*, vol. 12, pp. 45-57, 2009.

[11] G. Khanna; K. Beaty; G. Kar; A. Kochut, "Application Performance Management in Virtualized Server Environments," *Network Operations and Management Symposium*, NOMS 2006. pp.373-381, 2006

[12] D. Kusic, J.O. Kephart, J.E. Hanson, N. Kandasamy, G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster Computing*, vol. 12, no. 1, pp. 1-15

[13] P. Padala, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, "*Adaptive control of virtualized resources in utility computing environments,*" SIGOPS Operating Systems, rev. 41, 3, pp. 289-302

[14] L. Epstein, T. Tassa, "Vector assignment problems: A general framework", *Journal of Algorithms*, vol 48, pp 360-384, 2003.

[15] M. B. Reynolds, K. Hopkinson, M. Oxley, B. Mullins, "Provisioning Norm: An Asymmetric Quality Measure For SaaS Resource Allocation,", *8th IEEE International Conference on Services Computing*, pp. 112-119, 2011.

[16] M. B. Reynolds, D. Hulce, K. Hopkinson, M. Oxley, B. Mullins, "Cloud Chamber: A Self-Organizing Facility to Create, Exercise, and Examine Software as a Service Tenants,", *2012 45th Hawaii International Conference on System Sciences*, 2012.

[17] P. Manohar, A. Padmanath, S. Singh, D. Manjunath, "Multiperiod virtual topology design in wavelength routed optical networks. Circuits, Devices and Systems," IEE Proceedings, vol 150, no. 6. pp. 516-520, 2003.

[18] S. Albers, "Online algorithms: a survey," *Journal Mathematical Programming*, vol. 97, no. 1-2, pp. 3-26, 2003.

[19] R. Motwani, R. Prabhakar, *Random Algorithms*. Published by Cambridge University Press, 1995.

[20] Y. Bartal "The k-Server Problem – Lecture 11," Advanced Algorithms lecture notes. http://www.cs.huji.ac.il/~algo2/scribes/lecture9b.pdf.

[21] E. Grove, "The harmonic online k-server algorithm in competitive," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pp. 260-266, 1991.

[22] P. Raghavan, M. Snir, "Memory versus randomization in on-line algorithms," *IBM J. Res. Dev. 38*, pp. 683-707, 1994.

[23] A. Borodin, N. Linial, M.E. Saks, "An optimal on-line algorithm for metrical task system,". *J. ACM* vol. 39, no. 4 pp. 745-763, 1992.

[24] Y. Barta, B. Bollobás, "A Ramsey-Type Theorem for Metric Spaces and its Applications for Metrical Task Systems and Related Problems," *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, 2001.

[25] A. Fiat, M. Mendel, "Better algorithms for unfair metrical task systems and applications," *Proceedings of the Thirty-Second Annual ACM Symposium on theory of Computing*, pp. 725-734, 2000.

[26] *Capability Maturity Model Integration*, Carnegie Mellon, http://www.sei.cmu.edu/cmmi/

[27] B. Urgaonkar, P. Shenoy, T. Roscoe, “Resource overbooking and application profiling in a shared Internet hosting platform,” *ACM Trans. Internet Technology*, vol. 9, no. 1, pp. 1-45, 2009.

[28] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, A. Savva, “Job Submission Description Language (JSDL) Specification, Version 1.0,” *Open Grid Forum Technical Report*, vol. 28, http://www.ogf.org/documents/GFD.136.pdf. 2008.

[29] C.M. Jenkins, S.V. Rice, “A Typology for Resource Profiling and Modeling,” *Proceedings of the 40th Annual Simulation Symposium*, pp. 194-206, 2007.

[30] C. Stewart, K. Shen, “Performance modeling and system management for multi-component online services,” *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, vol. 2, pp. 71-84, 2005.

[31] M. Wooldridge, P.E. Dunne, “On the computational complexity of coalitional resource games,” *Artificial. Intelligence*, vol. 170, no. 10, pp. 835-871, 2006.

[32] H.C. Lau, L. Zhang, “Task Allocation via Multi-Agent Coalition Formation: Taxonomy, Algorithms and Complexity,” *Proceedings of the 15th IEEE international Conference on Tools with Artificial intelligence*, 2003.

[33] R.J. Gibbens, P.B. Key, “Coalition Games and Resource Allocation in Ad-Hoc Networks.” *Revised Selected Papers, Lecture Notes In Computer Science*, vol. 5151, pp. 387-398, 2008.

[34] T. Shrot, Y. Aumann, S. Kraus, “Easy and hard coalition resource game formation problems: a parameterized complexity analysis,” *Proceedings of the 8th international Conference on Autonomous Agents and Multiagent Systems*, vol. 1 pp. 433-440, 2009.

[35] N. Meuleau, K. Kim, L.P. Kaelbling,A.R. Cassandra, “Solving POMDPs by Searching the Space of Finite Policies,” *Proceedings of the Fifteenth International Conference on Uncertainty in Artificial Intelligence*, 1999.

[36] A.R. Cassandra, L.P. Kaelbling, M.L. Littman, “Acting optimally in partially observable stochastic domains,” *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[37] C.V. Goldman, “Optimizing information exchange in cooperative multi-agent systems, International Conference on Autonomous Agents,” *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pp. 137 – 144.

[38] D.A. Castanon, J.M. Wohletz, “Model predictive control for dynamic unreliable resource allocation,” *Proceedings of the 41st IEEE Conference on Decision and Control*, pp. 3754 - 3759, vol.4, 2002.

[39] M.M. Akbar, M.S. Rahman, M. Kaykobad, E.G. Manning, G.C. Shoja, “Solving the multidimensional multiple-choice knapsack problem by

constructing convex hulls," *Comput. Operation Research*, vol. 33, no. 5, pp. 1259-1273, 2006.

[40] M. Bartlett, A. Frisch, Y. Hamadi, I. Miguel, S.A. Tarim, M. Unsworth, "The Temporal Knapsack Problem and Its Solution," *LNCS* vol. 3524, pp. 34–48, 2005

[41] I. Alaya, C. Solnon, K. Gh´edira, "Ant Algorithm for the Multidimentional Knapsack Problem," *International Conference on Bioinspired Optimization Methods and their Applications*, 2004.

[42] D. Pisinger, "A minimal algorithm for the multiple-choice knapsack problem," *European Journal of Operational Research* vol. 83, pp. 394-410, 1995.

[43] D. Bertsimas R. Demir, "An approximate dynamic programming approach to multidimensional knapsack problems," *Management Science* vol. 48, pp. 550-565, 2002.

[44] B. Han, J. Leblet, G. Simon, "Hard multidimensional multiple choice knapsack problems, an empirical study," Comput. Operational. Research, vol. 37 no. 1, pp. 172-181, 2010.

[45] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, M. Trubian, "Resource Management in the Autonomic Service-Oriented Architecture," *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pp. 84-92, 2006.

[46] D. Ardagna, R. Mirandola, M. Trubian, L. Zhang, "Run-time resource management in SOA virtualized environments," *Proceedings of the 1st international Workshop on Quality of Service-Oriented Software Systems*, ACM, pp. 39-46, 2009.

[47] VMware, http://www.vmware.com/products/vsphere-hypervisor/ overview.html

[48] Tiny Core Linux, http://tinycorelinux.org/

[49] J. Poskanzer, http_load, ACME Laboratories, http://www.acme.com/ software/http_load/.

[50] D. Mosberger, T. Jin, "httperf - a tool for measuring web server performance," SIGMETRICS Performance Evaluation Review, vol. 26, iss. 3, pp. 31-37, doi:10.1145/306225.306235, http://www.hpl.hp.com/ research/linux/httperf/, 1998.

[51] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", The Internet Society, http://www.ietf.org/rfc/rfc2616.txt, 1999.

[52] P. Heinlein, FastCGI. *Linux Journal*, 55es, Article 1, 1998.

[53] fastCGI, http://www.fastcgi.com/.

[54] B. Adida, “It all starts at the server [5.World Wide Web and FastCGI],” *IEEE Internet Computing*, , vol.1, no.1, pp.75-77, 1997.

[55] G. E. P. Box, M. E. Muller, “A Note on the Generation of Random Normal Deviates,” *Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610-611, 1958.

[56] D. Chappell, “Introducing Windows Azure Platform,” Technical Report, David Chappell & Associates, http://www.davidchappell.com/writing/white_papers/Introducing_the_Windows_Azure_Platform,_v1.4--Chappell.pdf, 2010.

[57] M. B. Reynolds, K. Hopkinson, M. Oxley, B. Mullins, “Provisioning Norm: An Asymmetric Quality Measure For SaaS Resource Allocation,”, *8th IEEE International Conference on Services Computing*, pp. 112-119, 2011.

[58] J. Ferrer, V. Gregori, C. Alegre "Quasi-uniform Structures in Linear Lattices", *Journal: Rocky Mountain Journal of Mathematics*, vol. 23, no. 3, pp. 877-884, 1993.

[59] S. Chen; W. Li; S. Tian; Z. Mao, "On Optimization Problems in Quasi-Metric Spaces," *2006 International Conference on Machine Learning and Cybernetics*, pp.865-870, 13-16 Aug. 2006.

[60] D. Kincaid, W. Cheney, *Numerical Analysis: Mathematics of Scientific Computing*, American Mathematical Society, 3rd Revised edition, pp.93-110, 2002.

[61] M. B. Reynolds, K. Hopkinson, M. Oxley, B. Mullins, "Iterative Configuration Method: An Effective and Efficient Heuristic for Service Oriented Infrastructure Resource Allocation," *Proceedings of the 2010 6th World Congress on Services*, pp. 156-157, 2010.

[62] M. Resende, “Greedy Randomized Adaptive Search procedures (GRASP),” *AT&T Labs Research Technical Report: 98.41.1*, 1998.

[63] M. Resende, R. Werneck, “A Hybrid Multistart Heuristic for the Uncapacitated Facility Location Problem,” *European Journal of Operational Research*, vol. 174, issue 1, pp. 54-68, 2006.

[64] S. Kullback, R.A. Leibler, “On Information and Sufficiency,” *Annals of Mathematical Statistics*, vol. 22, pp. 79-86, 1951.

[65] T. Yu, Y. Zhang, K. Lin, “Efficient algorithms for Web services selection with end-to-end QoS constraints,” *ACM Transactions on the Web*, vol. 1, no. 1, 2007.

[66] R. Ricci, C. Alfeld, J. Lepreau, “A solver for the network testbed mapping problem,” *SIGCOMM Computer Communication*, vol. 33, num. 2, pp. 65-81, 2003.

[67] C. Hyser, B. Mckee, R. Gardner, B.J. Watson, “Autonomic virtual machine placement in the data center,” *HP Labs Technical Report HPL-2007-189*, 2007.

[68] J. Tang and M. Zhang, “An agent-based peer-to-peer grid computing architecture: convergence of grid and peer-to-peer computing”. *Proceedings of the 2006 Australasian Workshops on Grid Computing and E-Research*, pp. 33-39, 2006.

[69] X. Wang, Z. Du, Y. Chen, and S. Li, “Virtualization-based autonomic resource management for multi-tier Web applications in shared data center,” *Journal of Systems and Software*, vol. 81, pp. 1591-1608, 2008.

[70] Juszczyk, L.; Dustdar, S., “Script-Based Generation of Dynamic Testbeds for SOA,” *2010 IEEE International Conference on Web Services (ICWS)*, pp.195-202, 2010.

[71] L. Juszczyk, D. Schall, R. Mietzner, S. Dustdar, F. Leymann, “CAGE: Customizable Large-Scale SOA Testbeds in the Cloud,” *Lecture Notes in Computer Science Service-Oriented Computing*, vol. 6568, pp.76-87, 2011.

[72] D. Bianculli, W. Binder, and M. L. Drago. “Automated performance assessment for service-oriented middleware: a case study on BPEL engines,” *Proceedings of the 19th international conference on World wide web (WWW '10)*, pp. 141-150, 2010.

[73] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. “Model-Based Generation of Testbeds for Web Services,” *Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems: 8th International Workshop (TestCom '08 / FATES '08)*, pp. 266-282, 2008.

[74] X. Liu; J. Heo; L. Sha; X. Zhu; , “Queuing-Model-Based Adaptive Control of Multi-Tiered Web Applications,” *Network and Service Management, IEEE Transactions on*, vol.5, no.3, pp.157-167, 2008.

[75] Iqbal, W; Dailey, M. N.; Carrera, D. , “SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud,” Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on, pp.832-837, 17-20 May 2010

[76] The Transaction Processing Council, “TPC-W,” 2002, http://www.tpc.org/tpcw/default.asp.

[77] OW2 Consortium, “RUBiS: Rice University Bidding System,” 1999, http://rubis.ow2.org/

[78] A. Totok, V. Karamcheti, “Exploiting Service Usage Information for Optimizing Server Resource Management,” *ACM Trans. Internet Technology*, vol. 11, issue 1, 2011.

[79] D. Kempe, A. Dobra, J. Gehrke, “Gossip-based computation of aggregate information,” *Proc. 44th Annu. IEEE Symp. on Found. Of Comput. Sci.* , pp. 482–491, 2003.

[80] K. Jenkins, K. Hopkinson, K. Birman, “A gossip protocol for subgroup multicast,” *2001 International Conference on Distributed Computing Systems Workshop*, pp. 25-30, 2001.

[81] S, Sanghavi, B. Hajek, L. Massoulie, “Gossiping with Multiple Messages”, *INFOCOM 2007. 26th IEEE International Conference on Computer Communications*, pp.2135-2143, 2007.

[82] D. Knuth, “Seminumerical Algorithms”, *The Art of Computer Programming, Volume 2*, Addison Wesley, 1969.

[83] M.E. Crovella, A. Bestavros, “Self-similarity in World Wide Web traffic: evidence and possible causes,” *IEEE/ACM Transactions on Networking*, vol. 5, issue 6, pp. 835-846, 1997.

[84] H. Gupta, A. Mahanti, V.J. Ribeiro, “Revisiting coexistence of poissonity and self-similarity in Internet traffic,” *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2009*, pp. 1 - 10, 2009.

[85] A.J. Field, U. Harder, P.G. Harrison, “Measurement and modelling of self-similar traffic in computer networks,” *IEE Proceedings Communications*, vol. 151, issue 4, pp. 355-363, 2004.

[86] T.F. Abdelzaher, C. Lu, “Modeling and Performance Control of Internet Servers”, *Proceedings of the 39th IEEE Conference on Decision and Control*, 2000.

[87] C. Lu, T.F. Abdelzaber, J.A. Stankovic, S.H. Son, “A feedback control approach for guaranteeing relative delays in Web servers”, *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, 2001.

[88] T.F. Abdelzaher, K.G. Shin, N. Bhatti, “Performance Guarantees for Web Server End-Systems: A Control Theoretical Approach”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, January 2002.

[89] J.L. Hellerstein, Y. Diao, S. Parekh, D.M. Tilbury, *Feedback Control of Computing Systems*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[90] Y. Diao; N. Gandhi; J.L. Hellerstein; S. Parekh; D.M. Tilbury, “Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server,” *Network Operations and Management Symposium*, 2002.

[91] Y. Diao, J.L. Hellerstein, S. Parekh, “Optimizing Quality of Service Using Fuzzy Control”, *13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications (DSOM '02)*, 2002.

[92] J.L. Hellerstein, Y. Diao, S. Parekh, "A first-principles approach to constructing transfer functions for admission control in computing systems," *Proceedings of the 41st IEEE Conference on Decision and Control*, vol.3, no., pp. 2906- 2912, 2002.

[93] S. Parekh, “Managing the Performance Impact of Administrative Utilities”, IBM Research, Technical Report, http://www.research.ibm.com/PM/RC22864.pdf, 2003.

[94] S. Parekh, N. Gandhi, J.L. Hellerstein, D. Tilbury, T. Jayram, J. Bigus, "Using control theory to achieve service level objectives in performance management," *IEEE/IFIP International Symposium on Integrated Network Management Proceedings*, pp.841-854, 2001.

[95] J. Heo, P. Jayachandran, I. Shin, D. Wang, T. Abdelzaher, X. Liu, "OptiTuner: On Performance Composition and Server Farm Energy Minimization Application," *IEEE Transactions on Parallel and Distributed Systems*, vol.22, no.11, pp.1871-1878, 2011.

[96] M.A. Kjaer, M. Kihl, A. Robertsson, "Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers," *IEEE Transactions on Network and Service Management*, vol.6, no.4, pp.226-239, 2009.

[97] C. Xu, B. Liu, J. Wei, "Model Predictive Feedback Control for QoS Assurance in Webservers," *Computer*, vol.41, no.3, pp.66-72, 2008.

[98] L. Epstein, M. Levy, “Dynamic multi-dimensional bin packing”, *Journal of Discrete Algorithms,* vol 8, pp. 356-372, 2010.

[99] L. Epstein, E. Kleiman, “Self Bin Packing”, *Algorithmica*, vol 60, pp. 368-394, 2011.

[100]K. Ogata, *Modern Control Engineering, Fourth Edition*, Prentice Hall Publishers, pp 681-691, 2002.

[101]J.S. Milton, J.C. Arnold, *Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences*, fourth edition, McGraw Hill, pp. 352-357, 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)* 14 June 2012	**2. REPORT TYPE** Doctoral Dissertation	**3. DATES COVERED** *(From – To)* January 2009- June 2012

4. TITLE AND SUBTITLE Resource Provisioning in Large-Scale Self-Organizing Distributed Systems	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Reynolds, M. Brent, Mr.	**5d. PROJECT NUMBER** N/A
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENV) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/DCS/ENG/12-03
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally Left Blank	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

This dissertation researches the mathematical translation of resource provisioning policy into mathematical terms and parameters to solve the on-line service placement problem. A norm called the Provisioning Norm is introduced. Theorems presented in the work show the Provisioning Norm utility function and greedy, random, local search effectively and efficiently solve the on-line problem. Caching of placements is shown to reduce the cost of change but does not improve response time performance. The use of feedback control theory is shown to be effective at significantly improving performance but increases the cost of change. The theoretical results are verified using a decentralized, self-organizing testbed of web servers. The testbed places services on servers on-line using feedback control by profiling the service and node resources. Web servers share service profiles and find new service placement solutions using parallel searches based on the Provisioning Norm.

15. SUBJECT TERMS

Self-organizing, Provisioning Norm, Control Theory, Services, Virtual Machine, On-Line, Decentralized, Caching, Testbed

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Kenneth Hopkinson, Ph.D
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)*
U	U	U	UU	229	(937) 255-6565, x 4579 (Kenneth.hopkinson@afit.edu)